AFRL-OSR-VA-TR-2012-1229

Investigation of Large Scale Cortical Models on Clustered Multi-core Processors

Larry Dooley, Tarek M. Taha
University of Dayton

02-01-2013

Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/RTB1
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
02-01-2013

2. REPORT TYPE
Final Report

3. DATES COVERED *(From - To)*
03/2009-11/2011

4. TITLE AND SUBTITLE
Investigation of Large Scale Cortical Models on Clustered Multi-core Processors

5a. CONTRACT NUMBER
FA9550-09-1-0137

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Larry Dooley, Tarek M. Taha

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Dayton
300 College Park
Dayton, OH 45469

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR
875 N Randolph St
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution A: Approved for public release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Neuromorphic computing algorithms have become an area of strong interest for their strong inference capabilities. These algorithms are compute intensive and require high performance processing capabilities. This study examined the parallelization of several neuromorphic algorithms and their acceleration on a variety of highly parallel computing platforms. While the Bayesian algorithms examined had strong thread level parallelism, the neural algorithms examined had both data and thread level parallelism. As a result the Bayesian algorithms were mapped to chip-multiprocessors, such as Xeon processors, while the neural algorithms were mapped to both chip-multiprocessors and SIMD platforms, such as GPGPUs. Large compute clusters based on these processing architectures were also examined. The results indicate that these algorithms have a high degree of parallelism and are well suited multicore architectures. They are also well suited to large compute clusters of these multicore processors. In follow-on work, we are designing novel multicore neuromorphic computing architectures that will be several orders of magnitude more efficient than current systems.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:

a. REPORT	b. ABSTRACT	c. THIS PAGE

17. LIMITATION OF ABSTRACT

18. NUMBER OF PAGES

19a. NAME OF RESPONSIBLE PERSON

19b. TELEPHONE NUMBER *(Include area code)*

Reset

INSTRUCTIONS FOR COMPLETING SF 298

1. REPORT DATE. Full publication date, including day, month, if available. Must cite at least the year and be Year 2000 compliant, e.g. 30-06-1998; xx-06-1998; xx-xx-1998.

2. REPORT TYPE. State the type of report, such as final, technical, interim, memorandum, master's thesis, progress, quarterly, research, special, group study, etc.

3. DATES COVERED. Indicate the time during which the work was performed and the report was written, e.g., Jun 1997 - Jun 1998; 1-10 Jun 1996; May - Nov 1998; Nov 1998.

4. TITLE. Enter title and subtitle with volume number and part number, if applicable. On classified documents, enter the title classification in parentheses.

5a. CONTRACT NUMBER. Enter all contract numbers as they appear in the report, e.g. F33615-86-C-5169.

5b. GRANT NUMBER. Enter all grant numbers as they appear in the report, e.g. AFOSR-82-1234.

5c. PROGRAM ELEMENT NUMBER. Enter all program element numbers as they appear in the report, e.g. 61101A.

5d. PROJECT NUMBER. Enter all project numbers as they appear in the report, e.g. 1F665702D1257; ILIR.

5e. TASK NUMBER. Enter all task numbers as they appear in the report, e.g. 05; RF0330201; T4112.

5f. WORK UNIT NUMBER. Enter all work unit numbers as they appear in the report, e.g. 001; AFAPL30480105.

6. AUTHOR(S). Enter name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. The form of entry is the last name, first name, middle initial, and additional qualifiers separated by commas, e.g. Smith, Richard, J, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES). Self-explanatory.

8. PERFORMING ORGANIZATION REPORT NUMBER. Enter all unique alphanumeric report numbers assigned by the performing organization, e.g. BRL-1234; AFWL-TR-85-4017-Vol-21-PT-2.

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES). Enter the name and address of the organization(s) financially responsible for and monitoring the work.

10. SPONSOR/MONITOR'S ACRONYM(S). Enter, if available, e.g. BRL, ARDEC, NADC.

11. SPONSOR/MONITOR'S REPORT NUMBER(S). Enter report number as assigned by the sponsoring/ monitoring agency, if available, e.g. BRL-TR-829; -215.

12. DISTRIBUTION/AVAILABILITY STATEMENT. Use agency-mandated availability statements to indicate the public availability or distribution limitations of the report. If additional limitations/ restrictions or special markings are indicated, follow agency authorization procedures, e.g. RD/FRD, PROPIN, ITAR, etc. Include copyright information.

13. SUPPLEMENTARY NOTES. Enter information not included elsewhere such as: prepared in cooperation with; translation of; report supersedes; old edition number, etc.

14. ABSTRACT. A brief (approximately 200 words) factual summary of the most significant information.

15. SUBJECT TERMS. Key words or phrases identifying major concepts in the report.

16. SECURITY CLASSIFICATION. Enter security classification in accordance with security classification regulations, e.g. U, C, S, etc. If this form contains classified information, stamp classification level on the top and bottom of this page.

17. LIMITATION OF ABSTRACT. This block must be completed to assign a distribution limitation to the abstract. Enter UU (Unclassified Unlimited) or SAR (Same as Report). An entry in this block is necessary if the abstract is to be limited.

DISTRIBUTION A: Distribution approved for public release.

Investigation of Large Scale Cortical Models on Clustered Multi-core Processors

Final Report

AFOSR Grant FA9550-09-1-0137

PI: Dr. Tarek Taha (former), Dr. Larry Dooley (current)

Report Author: Dr. Tarek Taha (Associate Professor, Electrical Engineering, University of Dayton)

Date: February 1, 2013

Abstract

Neuromorphic computing algorithms have become an area of strong interest for their strong inference capabilities. These algorithms are compute intensive and require high performance processing capabilities. This study examined the parallelization of several neuromorphic algorithms and their acceleration on a variety of highly parallel computing platforms. While the Bayesian algorithms examined had strong thread level parallelism, the neural algorithms examined had both data and thread level parallelism. As a result the Bayesian algorithms were mapped to chip-multiprocessors, such as Xeon processors, while the neural algorithms were mapped to both chip-multiprocessors and SIMD platforms, such as GPGPUs. Large compute clusters based on these processing architectures were also examined.

The results indicate that these algorithms have a high degree of parallelism and are well suited multicore architectures. They are also well suited to large compute clusters of these multicore processors. In follow-on work, we are designing novel multicore neuromorphic computing architectures that will be several orders of magnitude more efficient than the architectures examined in this study. That work is based on the algorithm properties exposed in this study along with the computing architecture features we have found best suited for the algorithms.

1. Introduction

The human brain can perform more complex cognitive tasks at a faster rate than silicon based processors, despite the fact that neurons are much slower than the transistors used to design the processors. This is primarily because of the massive parallel processing employed in the neocortex. The main part of the brain dealing with learning and cognition is the neocortex. This is the outer layer of the human brain and is approximately the size of a large unfolded dinner napkin. It is estimated to consist of approximately 10^{11} neurons and 10^{14} connections between the neurons. Each neuron is connected to a large set of other neurons through extensions called dendrites and axons. Neurons communicate with each other by sending electrical pulses. These pulses are generated by the exchange of ions between the neurons.

There has been a strong interest amongst researchers to develop large parallel implementations of neuron models on the order of animal or human brains. At this scale, the models have the potential to provide much stronger inference capabilities than current generation computing algorithms [1]. A large domain of applications would benefit from the stronger inference capabilities including speech recognition, computer vision, textual and image content recognition, robotic control, and making sense of massive quantities of data.

Large scale models however require significant computing power to implement. Lansner et. al. [2] have shown that mouse sized cortical models developed on a cluster of commodity computers are computationally bound rather than communication bound. Thus the acceleration of neuron models on modern multicore architectures can provide significant benefits for the development of large scale cortical models. Multicore processors are the norm in the computing industry now given that it is difficult to increase the performance gains of single core processors (primarily because we have reached the limits on frequency scaling). Several research groups are examining the acceleration of neocortex models on commercial and custom multicore architectures.

Several mathematical models of processing within the neocortex have been proposed at different levels. The lowest level of modeling considers the chemical changes inside the soma, axon and dendrites of a neuron [3,4]. These models have significant complexity and provide insights into how neurons work. However they are too complex to develop large scale models of the brain as the computations for a single neuron can be computationally challenging.

The next level in the modeling hierarchy is based on modeling individual neurons using a set of differential equations. These artificial neural network models were first developed in 1952 [5]. Spiking neural networks fall under the third generation of these models and are currently in wide use [6]. They are significantly more biologically accurate than neural networks of the first two generations. Several large scale models based on spiking neural networks include [7-10,11].

Neuroanatomits have identified that a collection of about 80 to 100 neurons form into regular patterns of local cells running perpendicular to the cortical plane [12]. These collections of neurons are called mini-columns. Mountcastle [13] states that the basic unit of cortical operation is the mini-column and that a collection of mini-columns are grouped into a cortical column. He also states that the mini-columns within a cortical column are bound together by a common set of inputs and short-range horizontal connections.

Recently, several models of the neocortex have been proposed that are based on modeling mini-columns/columns. Modeling the neocortex based on cortical columns instead of individual neurons has several computational advantages: the number of nodes to model is decreased significantly (a node could represent a neuron or cortical column) and the number of connections between the nodes is sparse.

In fact, anatomical evidence [14] suggests that the connectivity between columns has the properties of a small-world graph [15], thus also enabling low latency communications between any two cortical columns [16]. Several mathematical models of the neocortex have been proposed based on cortical columns [17-21]. However, a set of recently published models based on cortical columns [19-21] provide significantly more insight into the workings of the neocortex. These newer models are based on hierarchical graphical networks and concur well with experimental results. They describe the brain as a hierarchical device that computes by performing sophisticated pattern matching and sequence prediction.

In this project we have examined the acceleration of neocortex models on high performance computing hardware. Several models representing different levels of complexity were chosen to examine the impact of performance. At the low level, several biologically accurate spiking neural network models were chosen. At a higher level, cortical column models, such as the Hierarchical Temporal Memory (HTM) model were chosen [22]. At an even high level of abstraction, the Cellular Simultaneous Recurrent Neural network (CSRN) model was selected for its ability to capture invariance [23]. The acceleration of several applications of neural models were examined, including face recognition, maze traversal, and robotic motion calibration. Radial Basis Function Neural Networks (RBFNN) were utilized for the robot motion calibration [24].

Several high performance multicore computing hardware were also selected in this study. These include IBM Cell processors, x86 processors (Intel Xeon and AMD Opteron), and NVIDIA GPGPUs. High performance computing clusters used in this study include the Air Force Research Lab Condor Cluster, the Ohio Super Computing Center Glenn Cluster, and the National Center for Supercomputing Applications Accelerator Cluster. Table 1 lists the platforms examined for acceleration of the different models.

Table 1. Acceleration platforms examined.

	x86	Cell	GPGPU
HTM [22]	×	×	
Dean [25]	×	×	
Izhikevich [26]	×	×	×
Hodgkin-Huxley [27]	×	×	×
Morris Lecar [28]	×	×	×
Wilson [29]	×	×	×
CSRN [23]	×		×
RBFNN [24]	×		×

2. Models examined

2.1 Spiking Neural Network Models

Spiking neural models capture neuronal behavior more accurately than traditional neural models. A neuron consists of three functionally distinct parts called dendrites, axons, and a soma. Each neuron is typically connected to over 10,000 other neurons [30]. The dendrites of a neuron collect input signals from other neurons, while the axons send output signals to other neurons. Input signals coming in along dendrites can cause changes in the ionic levels within the soma, which in turn can cause the neuron's membrane potential to change. If this membrane potential crosses a certain threshold, the neuron is said to have "fired" or "spiked". In these events the membrane potential rises rapidly for a short period of time (a spike) and causes electrical signals to be transmitted along the axons of the neuron to other neurons connected to it [31]. Spiking is the primary mechanism by which neurons send signals to each other. Over the last 50 years, several models have been proposed that capture the spiking mechanism within a neuron.

In this study, four of the more biologically accurate spiking neuron models (as listed by Izhikevich [32]) are examined. These are the Hodgkin-Huxley [27], Izhikevich [26], Wilson [29], and Morris-Lecar [28] models. The Hodgkin–Huxley model is considered to be one of the most biologically accurate spiking neuron models. All four of the models can reproduce almost all types of neuron responses that are seen in biological experiments. All but the Izhikevich model are based on biologically meaningful parameters (such as activation of Na and K currents, and inactivation of Na currents). Table 2 compares the computation properties of the four models. The Hodgkin–Huxley model utilizes exponential functions, while the Morris-Lecar model uses hyperbolic functions. These contribute to the higher flops needed for these two models.

Table 2. Spiking Network Properties

Model	Differential Equations	Variables updated each cycle	Other variables	Constants	Flops / neuron (Euler)	Flops / neuron (Runge-Kutta)
Izhikevich	2	2	2	4	13	70
Wilson	4	7	2	11	38	152
Morris-Lecar	2	5	2	12	147	297
Hodgkin-Huxley	4	16	2	10	246	442

Two common methods to implement the differential equations in these models include the Euler and the Runge-Kutta approaches. While the Runge-Kutta approach provides more accurate results, the Euler method is the most common approach for implementing the differential equations as it has a significantly lower computational load. In this study we primarily utilize the Euler approach, although we do examine the Runge-Kutta approach as well. The flop counts for both the Euler and the Runge-Kutta approach are listed in Table 2. These values are based on our implementations of the four models.

2.1.1 Izhikevich Model

Izhikevich proposed a new spiking neuron model in 2003 [26] that is based on only two differential equations (equations (1) and (2)). This model requires the least computations of all the models examined, because it needs fewer flops per neuron update and requires fewer neuron updates to be carried out per simulation run time (since the simulation time step is higher). However the model can still reproduce almost all types of neuron responses that are seen in biological experiments. The four constant parameters (*a*, *b*, *c* and *d*) can be initialized differently to allow modeling of various neural responses. A time step of 1 ms was utilized (as was done by Izhikevich in [26]).

$$\frac{dV}{dt} = 0.04V^2 + 5V + 140 - u + I \qquad (1)$$

$$\frac{du}{dt} = a(bV - u) \qquad (2)$$

$$\text{if } V \geq 30 \text{ mV, then } \begin{cases} V \leftarrow c \\ u \leftarrow u + d \end{cases}$$

2.1.2 Wilson Model

The Wilson model [29], proposed in 1999, requires four differential equations (equations (3) to (6)). The model has a large number of parameters than the Izhikevich model. Tuning these parameters allows the model to exhibit almost all neuronal properties. Three of the parameters in the differential equations (T_∞, R_∞, and m_∞) also need to be evaluated each cycle, thus adding a set of three more equations. A time step of 0.01 ms was utilized to update the four differential equations.

$$\frac{dH}{dt} = (1/45)(-H + 3T) \qquad (3)$$

$$\frac{dT}{dt} = (1/14)(-T + T_\infty) \qquad (4)$$

$$\frac{dR}{dt} = (1/\tau_R)(-R + R_\infty) \qquad (5)$$

$$\frac{dV}{dt} = (\frac{1}{C})(-m_\infty(V - E_{Na}) - 26R(V + E_K) - g_T T(V - E_{Ca}) - g_H H(V + E_K) + I) \qquad (6)$$

2.1.3 Morris-Lecar Model

Cathy Morris and Harold Lecar proposed a two dimensional conductance-based spiking model in 1981 [28]. The model consists of two differential equations (equations (7) and (8)). Three of the parameters in the differential equations (m_∞, w_∞, and τ_w) also need to be evaluated each cycle, thus adding a set of three more equations. These three equations involve hyperbolic functions, thus making it computationally more expensive than the Izhikevich and Wilson models. This computational load is lower than the Hodgkin-Huxley model however, thus making it popular in neurocomputation communities. A time step of 0.01 ms was utilized to update the two differential equations.

$$\frac{dv}{dt} = (\frac{1}{C})(I - g_{Ca} m_\infty (V - V_{Ca}) - g_K w(V - V_K) - g_{Leak}(V - V_{Leak})) \qquad (7)$$

$$\frac{dw}{dt} = (\frac{1}{\tau_w})(w_\infty - w)\phi \qquad (8)$$

2.1.4 Hodgkin Huxley Model

The Hodgkin–Huxley model [27] was a seminal work in neuron modeling. It consists of four differential equations (equations (9) to (12)). A set of 10 more equations have to be evaluated each cycle to update parameters used in the differential equations. Four of these equations involve exponential functions, thus making the model the most complex of the four models studied. A time step of 0.01 ms was utilized to update the four differential equations as this is the most commonly used value. This model is used in the detailed large scale neural simulations being carried out by IBM and EPFL [33].

$$\frac{dv}{dt} = (\frac{1}{C})\{I - g_K n^4(V - E_K) - g_{Na} m^3 h(V - E_{Na}) - g_L(V - E_L)\} \qquad (9)$$

$$\frac{dn}{dt} = (n_\infty(V) - n) / \tau_n(V) \tag{10}$$

$$\frac{dm}{dt} = (m_\infty(V) - m) / \tau_m(V) \tag{11}$$

$$\frac{dh}{dt} = (h_\infty(V) - h) / \tau_h(V) \tag{12}$$

2.1.5 Difference equation solution: Runge-Kutte versus Euler

The Runge-Kutte method is a numerical solution algorithm for difference equations. It provides more precise results when compared with ordinary Euler algorithms. Precise of simulation is very important if the network has a recurrent structure. The recurrent structure is very common in real neural networks on brain's cortex level. In the Euler algorithm, we simply use the current status as the difference and multiply the value with a time interval to calculate the increment of state. The Runge-Kutta method uses a combination of multiple calculations to eliminate the high-order error terms. For example, if we need to examine the equation:

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{13}$$

In four-order RK algorithm, we divide the state update process into four steps as follow:

$$\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\
k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\
k_4 &= hf(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\end{aligned} \tag{14}$$

It is clear that because we divide the update of difference equation into several steps, the RK algorithm requires more FLOPs to compute the state of neuron. Then difference of FLOPs between Euler and RK simulation is shown in Table 2.

2.2 Hierarchical Temporal Memory Model

George and Hawkins developed an initial mathematical model [34] of the neocortex based on the framework described by Hawkins in [35]. Their model utilizes a hierarchical collection of nodes that employ Pearl's Bayesian belief propagation algorithm [36]. As shown in Figure 1, each node has one parent and multiple children. Input data is fed into the bottom layer of nodes (level 1) after undergoing some preprocessing. After a set of feed-forward and feedback belief propagations between nodes in the network, a final belief is available at the top level node. This belief is a distribution that indicates the degree of similarity between the input and the different items the network has been trained to recognize. The model is trained in a supervised manner by presenting the training data multiple times to the bottom layer of nodes.

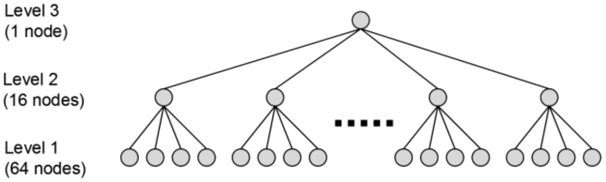


Figure 1. Network structure of HTM model implemented.

The computational algorithm within each node of the model is identical and follows equations (15) through (20) below. The nodes send belief vectors to each other (π and λ) and utilize an internal probability matrix, P_{xu} (generated in an offline training phase).

$$\lambda_{product}[i] = \prod_{child} \lambda_{in}[child][i] \tag{15}$$

$$F_{xu}[j][k] = \pi_{in}[j] \times P_{xu}[j][k] \times \lambda_{product}[k] \tag{16}$$

$$m_{row}[j] = max(m_{row}[j], F_{xu}[j][k]) \tag{17}$$

$$m_{col}[k] = max(m_{col}[k], F_{xu}[j][k]) \tag{18}$$

$$\lambda_{out}[j] = m_{row}[j] / \pi_{in}[j] \tag{19}$$

$$\pi_{out}[child][k] = m_{col}[k] / \lambda_{in}[child][k] \tag{20}$$

2.3 Dean Model

Thomas Dean proposed a hierarchical Bayesian model [37] based on the work by Lee and Mumford [38] to model the invariant pattern recognition seen in the visual cortex. The example model examined in this study consists of a hierarchy of nodes with each node connected to a set of lower level nodes. There is a degree of overlap in the receptive field of the nodes in some of the layers (such as layer 2 in Figure 2). Inputs to the layer 1 nodes are processed through a set of feed-forward and feedback processing steps through the network. A final inference based on this input is produced by the top layer node. The model is trained in a supervised manner by presenting a set of training data to the bottom layer of nodes multiple times.

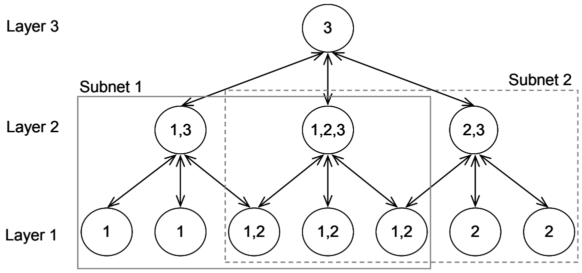


Figure 2. A simple example of Thomas Dean's hierarchical Bayesian network model. This network can be divided into three subnets as shown. The nodes are numbered with the subnets they belong to.

Dean examined several approaches to process the hierarchical Bayesian network structure. The approach examined in this study is the one proposed by Dean where the network is decomposed into a set of subnets, and each subnet is evaluated individually. This decomposes the full tree into multiple subcomponents, thus simplifying the overall evaluation. A subnet can be defined as a node, its parents, and all the children of those parents in the same level as the original node, and as shown in Figure 2, a node can belong to multiple subnets. Each subnet produces evidence to send to the next layer of subnets.

Algorithm 1. Processing in the Dean model

```
Preprocess inputs: find mixture of Gaussian for each 4×4 pixel patch
Repeat till output convergence:
   Upward pass (from layers 1 to 3):
      For all subnets in a layer:
         Incorporate evidence from below (get image evidence or lambda values)
         Process junction tree (collecting and distributing evidence)
         Calculate evidence to send to upper layer of subnets (lambda values)
   Downward pass (from layers 3 to 1):
      For all subnets in a layer:
         Incorporate evidence from above (get pi values)
         Process junction tree (collecting and distributing evidence)
         Calculate evidence to send to lower layer of subnets (pi values)
Read output
```

In order to process a subnet, it is first converted to its equivalent junction-tree representation. The Lauritzen and Spiegelhalter's junction-tree algorithm [39] is utilized for exact inference in the tree. Algorithm 1 lists the set of steps involved in the recognition phase of the Dean model.

2.4 Cellular Simultaneous Recurrent Network Model

CSRNs are a recent class of biologically inspired algorithms that have several significant advantages over other neural algorithms for distortion invariant image recognition. Firstly, they are more capable than regular recurrent networks (RNNs), such as the Elman network, in capturing temporal information. Secondly, CSRNs combine the ideas in cellular neural networks (CNNs) with RNNs to drastically reduce the number of adjustable weights in the network. CSRNs have been proven more effective and flexible than intricately hand crafted solutions at addressing a wide range of challenging problems, such as path optimization for maze traversal [40] and distortion invariance in image recognition. The same cannot be said of traditional specialized image recognition algorithms – for instance traditional face recognition algorithms cannot be applied to optimization problems.

In [41], CSRNs were applied to pose invariant face recognition, a task where traditional computer vision methods underperform, and were shown to achieve an overall 77% face recognition rate using the VidTIMIT database. Although powerful in image processing capabilities, CSRNs have high computational demands with increasing input problem size. In order to process large databases, efficient processing approaches for implementing CSRNs need to be investigated.

In this work, the CSRN cell element structure for the face recognition used the generalized multi-layered perceptron (GMLP) model is shown in Figure 3. This GMLP model works in two layers. The first layer acts as an input layer. It is composed of a bias node, two external input nodes, four neighbor input nodes corresponding to up, down, left, and right cell neighbor outputs, and 10 recurrent nodes. The second layer acts as a hidden layer consisting of only the recurrent nodes.

The nodes are fully connected between the first and second layer. Those connections are also weighted with the weights associated with the bias node (gray) denoted as *ww* and the weights associated with the remaining first layer nodes (black) denoted as *W*. In forming the overall network cell element output, the second layer node output values are also aggregated as input from one node to all of the succeeding nodes. Ultimately, the last second layer node will receive all preceding second layer node outputs multiplied by a weight from *W* along with the weighted outputs from the first layer nodes as input. The output of the last second layer node is multiplied by a weight scaling value Ws and that is observed as the output of that particular CSRN cell element.

Ren et al.[42] mapped a single CSRN to a single pattern vector component. The CSRN processes inputs as a 2D grid of a pattern vector component's temporal signature values. Each CSRN cell element receives a different temporal signature value as input from that pattern vector. Therefore, if a face sequence has been sampled 10 times resulting in nine temporal signatures, then a 3×3 grid of the temporal signature data corresponding to one component in the pattern vector is submitted to a CSRN in the network for processing.

In the case of this application, each CSRN cell element uses a 27 node GMLP model. The GMLP model has two layers where the first layer consisted of 17 nodes (one bias, two external inputs, four neighbor nodes, and 10 recurrent nodes) and the second layer consisted of 10 nodes (10 recurrent nodes). The bias node feeds a constant value of one into the input layer. The first of the two external inputs become the value of the temporal signature pattern vector. The second external input is set to one whenever the pattern vector value equals zero and is otherwise set to zero.

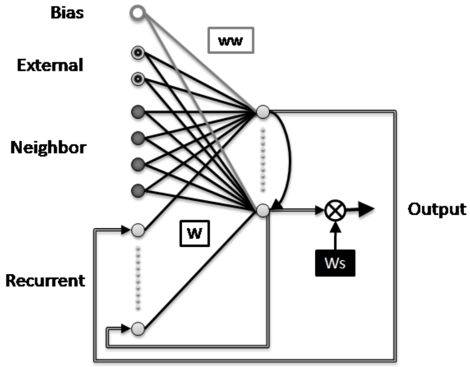


Figure 3. Two layer GMLP network. One GMLP network is used for each cell in the CSRN. Nodes are fully connected between layers.

CSRNs can be trained several ways, but the method which Ilin et al. [40] observed to have the best results was multi-stream extended Kalman filter (MSEKF) training technique. MSEKF works mainly using the following four equations:

$$\Gamma_t = C_t K_t C_t^T + R_t \tag{21}$$

$$G_t = K_t C_t^T \Gamma_t^{-1} \tag{22}$$

$$\vec{w}_{t+1} = \vec{w}_t + G_t \alpha_t \tag{23}$$

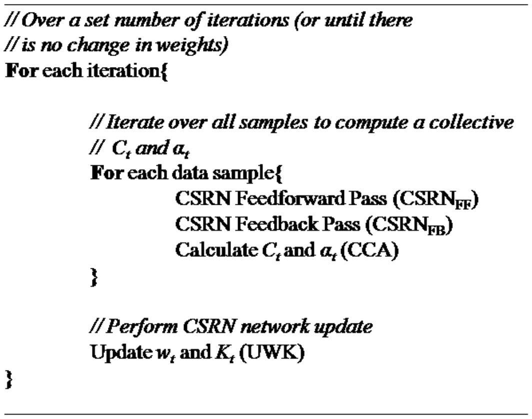
$$K_{t+1} = K_t - G_t C_t K_t + Q_t \tag{24}$$

This is where the variables for time iteration t represent the following: Γ_t is the residual covariance, C_t is the state observation matrix Jacobian, K_t is the predicted estimate covariance, R_t is the observation noise covariance, G_t is the optimal Kalman gain, α_t is the measurement residual, w_t is the predicted state, and Q_t is the process noise covariance. For training, w_t represents the shared weights for the CSRN (W and ww from Figure 3). Also, C_t and α_t are computed based upon w_t.

Training of a single CSRN using MSEKF for the face recognition problem can broken down into four stages as outlined in Algorithm 2. The CSRN feedforward pass (CSRN_{FF}) is for processing a data sample. During CSRN_{FF}, the data sample is propagated up through the GMLP contained within the CSRN cells. The output of CSRNFF is used as the overall output as well as to compute α_t. The CSRN feedback pass

(CSRN_{FB}) is used mainly for helping computing C_t. During CSRN_{FB}, the outputs of CSRN_{FF} are propagated back down through the GMLP contained within the CSRN cells. CSRN_{FF} and CSRN_{FB} both iterate over a predefined interval.

Algorithm 2. Pseudocode for CSRN MSEKF training.



```
// Over a set number of iterations (or until there
// is no change in weights)
For each iteration{

        // Iterate over all samples to compute a collective
        // $C_t$ and $\alpha_t$
        For each data sample{
                CSRN Feedforward Pass ($\mathrm{CSRN}_{FF}$)
                CSRN Feedback Pass ($\mathrm{CSRN}_{FB}$)
                Calculate $C_t$ and $\alpha_t$ (CCA)
        }

        // Perform CSRN network update
        Update $w_t$ and $K_t$ (UWK)
}
```

Once the outputs of both the CSRN_{FF} and CSRN_{FB} stage have been obtained, C_t and α_t can be computed (CCA). After C_t and α_t are computed, K_{t+1} and w_{t+1} can be computed (UWK in Figure 13) which consists of performing Equations (21) – (24). To train a network of CSRNs, this process must be done for all CSRNs within the network. Likewise for testing, a network of CSRNs only needs to perform the CSRN_{FF} stage for their respective data samples.

2.5 Radial Basis Function (RBF) neural networks

Robot calibration is an important method to improve robot accuracy. Robot calibration typically consists of determining the actual values of kinematic and dynamic parameters of the robot [43]. However, the design of some industrial robots and their controllers do not allow the user to change some of those calibration parameters.

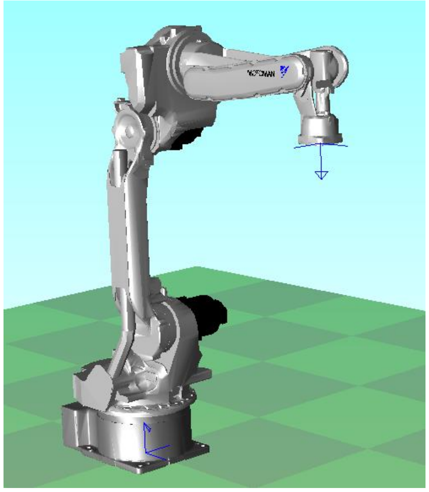


Figure 4. The MOTOMAN MA1400 Manipulator [44].

In this study, we introduce the notion of *filtering* as an effective means to calibrate an industrial manipulator. We utilize the MOTOMAN MA1400 manipulator and the NX 100 controller [44] for our experimental effort. Figure 4 depicts the MOTOMAN MA1400 manipulator. The MA1400 manipulator has six revolute joints (i.e., 6-DOF manipulator). Our method makes use of a Radial Basis Function (RBF) based neural network for calibration as shown in Figure 5. The RBF network is used to find the functional relationship between *actual or measured* position data points (physical position coordinates of the end-effector deduced using an external measuring device) and *desired* position data points. Note that the discrepancy between the *actual* and *desired* position data points is due to de-calibration. Our approach shows an accuracy/precision improvement of about 88%.

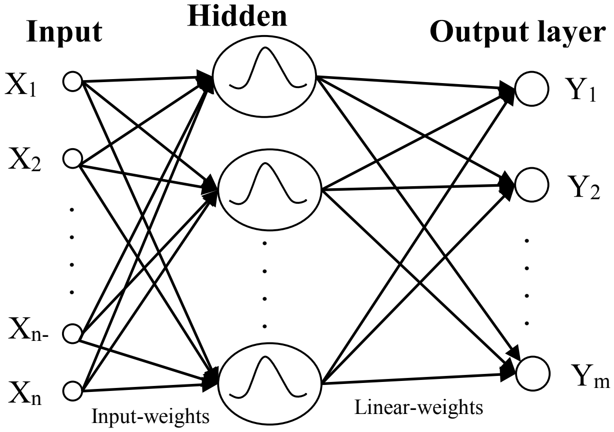


Figure 5. The RBF neural network structure.

3. Acceleration

3.1 Spiking Neural Networks

Four versions of an image recognition network were developed corresponding to the four spiking models studied (see Algorithm 3). The main between these versions difference was in the equations utilized to update the potential of the neurons. For each model, two subversions were developed – one for Euler version of the update equations and one for the Runge-Kutta version. The network consisted of two layers, where the first layer acted as input neurons and the second layer as output neurons. Input images were presented to the first layer of neurons, with each image pixel corresponding to a separate input neuron. Thus the number of neurons in the first layer is equal to the number of pixels in the input image. Binary input images were utilized in this study. The number of output neurons was equal to the number of training images. Each input neuron was connected to all the output neurons. A prototype of this network is shown in Figure 6. A set of 48 training images were developed to train the network (see Figure 7).

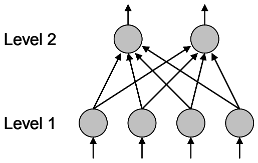


Figure 6. Network used for testing spiking models.

Algorithm 3. The testing phase of the spiking neuron image recognition model

```
Repeat till a level two neuron fires:
    For all level one neurons:
        Read input current
        Calculate neuron membrane voltage
        If neuron fires, upgrade the level 2 input current
—Barrier—
    For all level two neurons:
        For each non zero number of firing from level one (from previous cycle),
            calculate total level 2 input current
        Calculate neuron membrane voltage
            If neuron fires, output is produced
—Barrier—
```

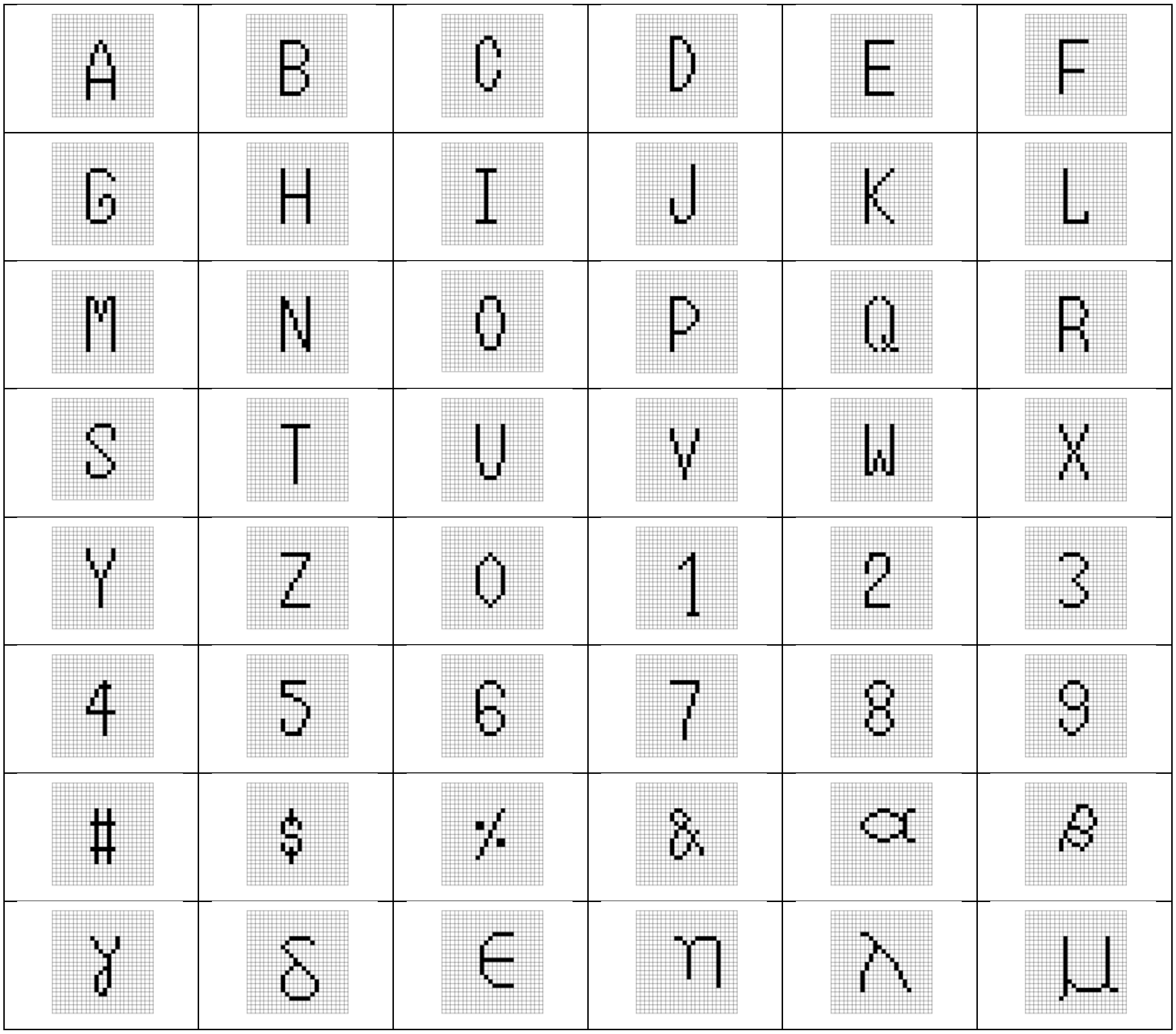


Figure 7. Training images utilized. There are a total of 48 24×24 pixel images.

The networks were scaled to large sizes for testing on the different platforms. In this scaling, the number of level 1 layer neurons was increased, while the number of level 2 neurons were held constant. The size of the input images were scaled with the number of level 1 neurons. Table 3 shows the networks tested.

Table 3. Networks implemented on the Opteron cluster.

Size of input image	Level 1 neurons	Level2 neurons	Synapses
3072×3072	9,437,184	48	452,984,832
6144×6144	37,748,736	48	1,811,939,328
9216×9216	84,934,656	48	4,076,863,488
12288×12288	150,798,400	48	7,238,323,200
15360×15360	235,929,600	48	11,324,620,800

The algorithms were parallelized by splitting the level 1 neurons over each of the cores available. The Cell processor requires some specific code optimizations to achieve high performance. The code optimizations described for the Opteron processors (vectorization, and minimizing thread creation) are essential to obtain high performance on the Cell architecture. Several other explicit code changes are needed as well, including branch elimination and double buffering. Since the SPEs in the Cell processor do not contain any branch prediction units, it is essential to reduce the number of branch instructions in the code run on the SPEs. This was accomplished by unrolling loops and in-lining most function calls. Given that data transfers to the SPEs on the Cell are explicitly programmer controlled, double buffering was needed to ensure that data transfers took place in parallel with data evaluation. In this process, once the data required for the first iteration of a loop has been transferred, the first iteration of the loop can be evaluated simultaneously with the DMA data transfer for the second iteration of the loop. Mailboxes were used for synchronization between SPEs. In case of the GPGPU implementation, each neuron was processed as a separate thread. Figure 8 shows the steps involved in the execution of the network on a cluster of GPUS.

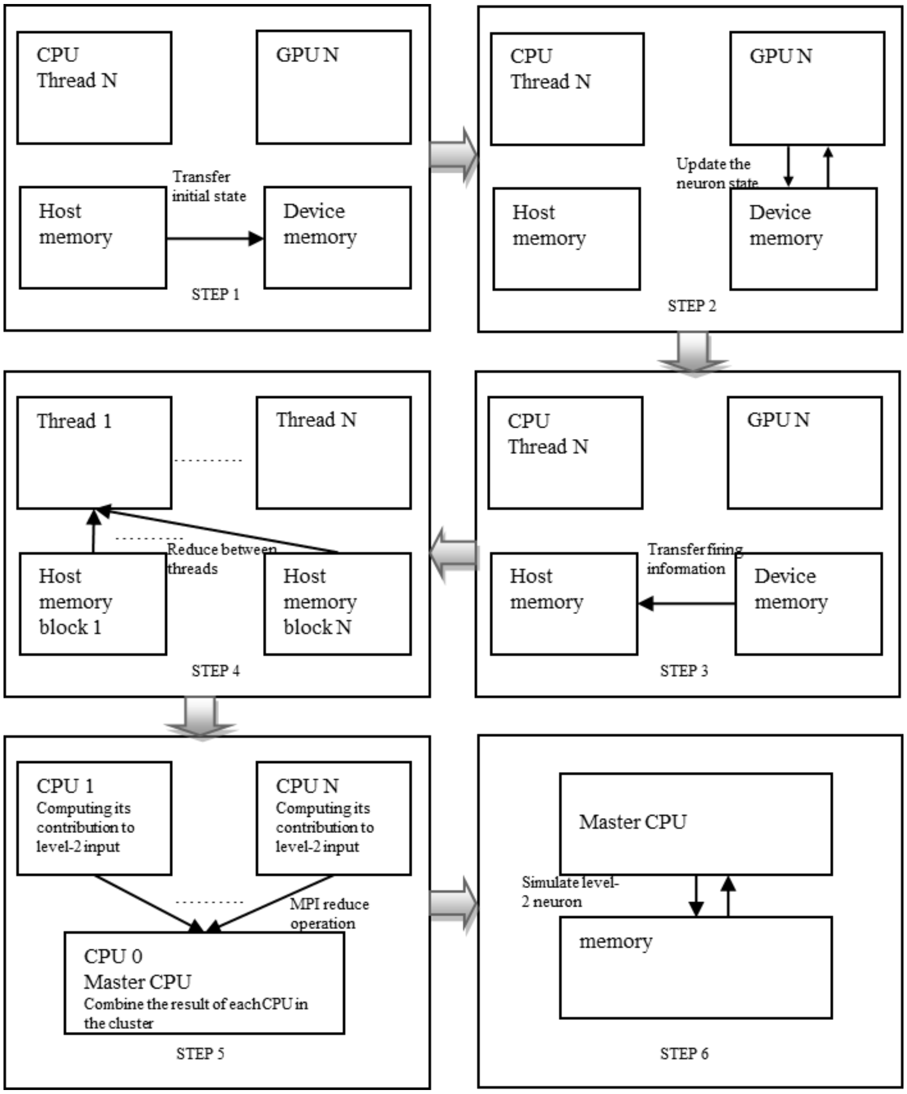


Figure 8. Steps involved in the execution of the spiking neural network on a cluster of GPUS.

3.2 HTM Model

3.2.1 Network parallelization

All the nodes in a particular layer are independent of each other and can therefore be evaluated in parallel. Therefore in this study, the HTM network was parallelized by assigning groups of nodes in a particular layer to separate processing cores. Nearly all computations in equations 15 through 20 are element-by-element matrix multiplies and divides (thus there are no addition operations needed). In order to accelerate the computations, the matrix values were converted into logarithmic form so that more expensive multiplies and divides could be replaced by less time consuming additions and subtractions. The comparisons involved in equations 17 and 18 could still be performed in logarithmic form and were thus unaffected by this change.

3.2.2 P_{xu} matrix compression and model vectorization

The P_{xu} matrix in equation 16 is large enough that it needs special consideration when examining the vectorization of the nodes. These matrices themselves are extremely sparse, being made up almost 90% zeroes. The computations in equations 15 through 20 are element-by-element rather than dot products. Compressing the P_{xu} matrices can significantly speed up the algorithm computation by skipping over strings of zeros. Thus any vectorization approach needs to consider the compression of the P_{xu} matrix. Two possible approaches to utilize vectorization for the George Hawkins model were examined. The first involves vectorizing the code to process a single image more efficiently. The second approach involves vectorizing the code to process multiple images simultaneously.

Single image vectorization: In this case, equations 15 through 20 need to be vectorized for a single image. Equations 15, 19, and 20 can be vectorized easily if the variables for the equations are padded to be multiples of the vector width. Equation 16, however, cannot be vectorized as easily, given that the P_{xu} matrix is sparse. We examined the feasibility of block compression [45] of the P_{xu} matrix to vectorize the computations in equation 16. In order to be efficient, there should be on high density of non zero elements in uncompressed blocks.

Two possible approaches for block compression are to compress along the rows or along the columns of the target matrix. Tables 4 and 5 show the density of non zero blocks for both row and column wise compression with block sizes of 4 and 8 respectively. Several network sizes are examined. The results indicate that with a vectorization factor of four, the average P_{xu} uncompressed block contains less than two non-zero elements per block, while a vectorization factor of eight yields at most 2 elements per block on average. Thus vectorizing the equations for single images is not very efficient.

Table 4. Block compression of P_{xu} with a block size of 4.

	Compression along rows			Compression along columns		
Network Size	Percentage of non-zero blocks	Average non-zero elements in non-zero blocks	Percentage of non-zero blocks with more than two non-zero elements	Percentage of non-zero blocks	Average non-zero elements in non-zero blocks	Percentage of non-zero blocks with more than two non-zero elements
81	*3.84*	*1.2531*	*20.06*	*3.97*	*1.2605*	*17.72*
181	*5.17*	*1.3035*	*22.82*	*5.02*	*1.3891*	*24.89*
321	*6.23*	*1.3935*	*27.47*	*5.96*	*1.4940*	*28.79*
501	*7.41*	*1.4624*	*30.53*	*6.72*	*1.6483*	*34.48*
721	*7.82*	*1.4848*	*31.4*	*7.12*	*1.6659*	*35.02*

Table 5. Block compression of P_{xu} with a block size of 8.

	Compression along rows			Compression along columns		
Network Size	Percentage of non-zero blocks	Average non-zero elements in non-zero blocks	Percentage of non-zero blocks with more than two non-zero elements	Percentage of non-zero blocks	Average non-zero elements in non-zero blocks	Percentage of non-zero blocks with more than two non-zero elements
81	*6.44*	*1.4587*	*27.89*	*6.97*	*1.4374*	*22.74*
181	*8.32*	*1.5837*	*32.66*	*8.38*	*1.6506*	*29.77*
321	*9.75*	*1.7418*	*36.69*	*9.73*	*1.8270*	*33.35*
501	*11.31*	*1.8754*	*39.91*	*10.75*	*2.0571*	*38.98*
721	*11.88*	*1.9125*	*40.38*	*11.36*	*2.0851*	*39.45*

Multiple image vectorization: The computations for any input image are identical throughout the network because each node in the network processes any input given in exactly the same manner. Therefore multiple images can also be evaluated in parallel using vectorization. In this case any compression scheme can be adopted for the P_{xu} matrices. We compress the matrix by providing a coordinate for each nonzero value in the P_{xu} matrix. Two approaches for dealing with this are to treat the P_{xu} matrix as a linear vector (see Figure 9(b)) or to treat it as a two dimensional matrix (see Figure 9(c)). In the former case, only one coordinate is needed per nonzero element, while in the latter case, two coordinate values are needed. The first approach results in a higher compression level and thus lower data transfer time. It however does require the generation of a two dimensional (x,y) coordinate for each linear coordinate (for equation 16). Our studies indicate that a two dimensional representation provides the lowest overall execution time. For example, for the 721 node HTM model, the single dimensional approach required 18.26 ms on a Playstation 3, while the two dimensional approach required 10.96 ms.

The two dimensional representation utilized is very similar to the Yale Sparse Matrix Format. The Yale format utilizes three vectors: a list of the non-zero elements (A), the column index of each non-zero element (JA), and the index in vector A which correspond to the first non-zero element of each row (IA). In our case, the last vector is replaced with a row index of each element. The last vector of Yale format (IA) would make loop unrolling more difficult, as there would be extra overhead in calculating the row index needed in the calculations (from vector IA). Our tests show that the Yale format leads to at most 20% extra compression of the matrices compared to our approach, but increased runtime by 19% due to the overhead mentioned.

1	**0**	**2**	**0**	**0**
0	**4**	**0**	**0**	**0**
0	**6**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**9**

(a)

1	*0*	**2**	*2*	**4**
6	**6**	*11*	**1**	*15*
9	*24*			

(b)

1	*0*	*0*	**2**	*0*
2	**4**	*1*	*1*	**6**
1	*2*	**1**	*3*	*0*
9	*4*	*4*		

(c)

Figure 9. Restructuring the P_{xu} matrix. (a) Original P_{xu} Matrix. (b) Single dimensional position representation, [p :value, *x* :coordinate]. (c) Two dimensional position representation, [p :value, *x,y* : coordinates]

3.3 Dean Model

Algorithm 4 shows the overall set of steps in processing the Dean model. The first step is to preprocess the input image (line 1). For any given input image, the network is processed through multiple bottom-to-top (line 3) and top-to-bottom (line 8) passes. In each pass, all the subnets for a certain layer are processed before moving to the next layer. The process has to be repeated at least twice to check for output convergence (line 2). The output is generated by the top level subnet during the upward pass. The processing inside a subnet is similar for both the upward and downward passes (described by lines 5-7 for the upward pass and lines 10-12 for the downward pass).

In order to process a subnet, it is first converted to its equivalent junction-tree representation (see Figure 10). The subnet to junction tree mapping is carried out during training and does not have to be redone during inference (as the mapping is reused). A junction-tree consists of a set of nodes called cliques, where each clique is based on a collection of nodes in the original subnet. Each clique has a potential based on the conditional probability tables of the nodes in the subnet it is composed of. The connection between two cliques is called a separator and has a separator potential based on a reduced form of one of the clique potentials (with the clique chosen being the one sending information to the other). The operations in the junction tree processing consist primarily of element by element multi-dimensional matrix adds, multiplies, and divides.

Algorithm 4. Processing in the Dean model

```
 1.   Preprocess inputs: find mixture of Gaussian for each 4×4 pixel patch

 2.   Repeat till output convergence:
 3.       Upward pass (from layers 1 to 3):
 4.           For all subnets in a layer:
 5.               Incorporate evidence (from below) and initialize junction tree
 6.               Process junction tree (collecting and distributing evidence)
 7.               Calculate evidence to send to upper layer of subnets (lambda values)
 8.       Downward pass (from layers 3 to 1):
 9.           For all subnets in a layer:
10.               Incorporate evidence (from above) and initialize junction tree
11.               Process junction tree (collecting and distributing evidence)
12.               Calculate evidence to send to lower layer of subnets (pi values)

13.   Read output
```

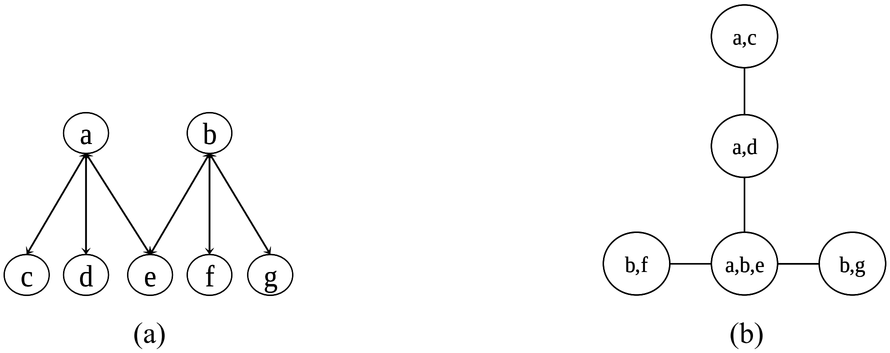


Figure 10. A subnet and its corresponding junction tree. Part (a) shows a subnet similar to the lowest level subnets in Figure 2 (nodes are labeled a through g). Part (b) shows the corresponding junction tree of this subnet. Cliques are labeled with the corresponding nodes in the subnet that they are composed of.

The processing inside a subnet takes place through the following three parts:

Part 1 (lines 5 and 10): Incorporate evidence and initialize junction tree. All the nodes in the network receive π or λ belief values (soft evidence) from the subnets above or below them respectively. All the λ values are initialized to one. Additionally, the bottom layer nodes see the preprocessed input image as hard evidence (E). The junction tree corresponding to a subnet has its clique potentials (ψ_C) initialized based on the subnet node potentials (P(X$_i$)) and the input evidence:

$$\psi_C = \prod_i (P(X_i|E) \times \lambda_i \times k_{C,i}), \text{ where } k_{C,i} = 0 \text{ or } 1 \qquad (24)$$

The values of $k_{C,i}$ are determined through the training process (here C represents the clique being examined). A $k_{C,i}$ value of zero denotes that the potential of the node i in clique C should not be considered while calculating the clique potentials. In the downward pass (line 10), the potential of upper nodes in each subnet get their potentials replaced by the π belief received from above (P(X)= π_X).

Part 2 (lines 6 and 11): Process junction tree. The Lauritzen and Spiegelhalter's junction-tree algorithm [39] is utilized for exact inference in the tree. The junction tree derived from a subnet is evaluated in a single bottom-to-top (collect evidence) and then top-to-bottom pass (distribute evidence).

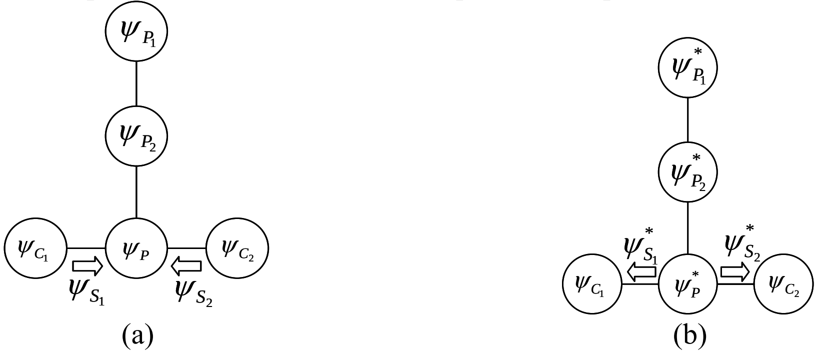


Figure 11. Processing in a junction tree using the Lauritzen Spiegelhalter algorithm. Part (a) shows the evidence being collected from children to a parent node. Part (b) shows the evidence being distributed back to the children after the parent has been updated.

In the collect evidence pass, separator potentials (ψ_{S_i}) are first calculated based on the child clique potentials (ψ_{C_i}) as shown in equation 25. The parent clique potential is then updated (ψ_P) based on the separator potentials (ψ_{S_i}) as shown in equation 26. This process is illustrated in Figure 11 (a) and continues recursively until the root is updated.

$$\psi_{S_i} = \sum_{C_i \backslash S_i} \psi_{C_i} \qquad (25)$$

$$\psi^*_P = \psi_P \times \prod_i \psi_{S_i} \quad (26)$$

The distribute evidence pass starts once the root clique in the junction tree has been updated through the collect evidence phase. In this pass, new separator potentials ($\psi^*_{S_i}$) are calculated by reducing the potentials of the parent clique (as shown in equation 27). The children are then updated using both the old and updated separator potentials (ψ_{S_i} and $\psi^*_{S_i}$ respectively as shown in equation 28). The updated values are propagated downward recursively until all leaf cliques are updated. The process is illustrated in Figure 11 (b).

$$\psi^*_{S_i} = \sum_{P \backslash S_i} \psi^*_P \qquad (27)$$

$$\psi^*_{C_i} = \psi^*_{S_i} \times (\psi_{C_i} / \psi_{S_i}) \quad (28)$$

Part 3 (lines 7 and 12): Calculate evidence to be sent. Evidence to be sent to the next layer of subnets is calculated in this phase. While going through the upward pass (line 7), these are the λ values, and while going down (line 12) these are the π values. Lambda values are updated based on equation 29. Here P(I) is a prior distribution of the training classes generated during the training process, while P(E) is the sum of all the clique potentials in the junction tree. During the downward pass, π values are generated based on equation 30.

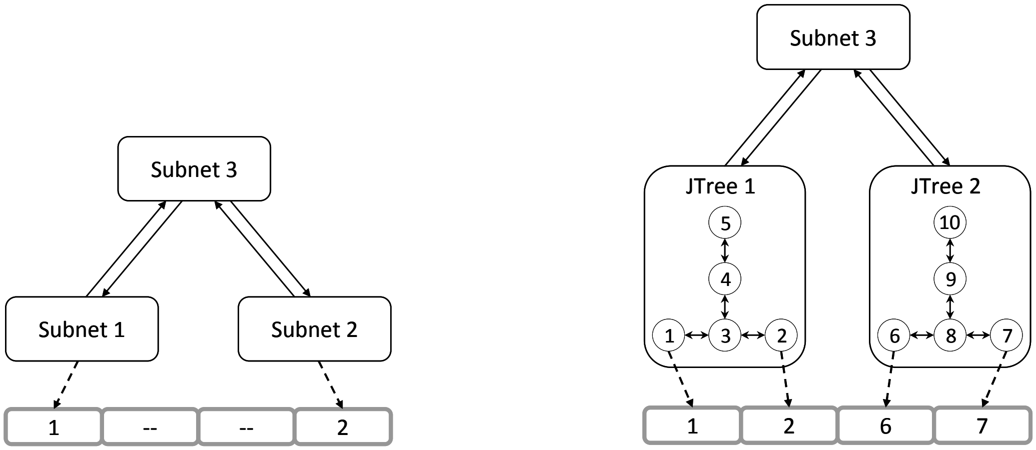
$$\text{if } k_{c,x} = 1,\ \lambda_X = \left(P(E) / P(I) \right) \times \sum_{X \setminus C} \psi_C \tag{29}$$

$$\text{if } k_{c,x} = 1,\ \pi_X = \sum_{X \setminus C} \psi_C \tag{30}$$

3.3.1 Network parallelization

The nodes in a network in the Dean model can be grouped into subnets and the network would be processed by evaluating subnets rather than individual nodes. Also, as shown in the same section, each subnet was evaluated by processing its junction-tree representation. Each node of the junction tree is called a clique and has a clique potential associated with it. This potential is derived by combining the conditional probability tables of each node in the subnet that forms the clique (these tables are multi-dimensional with a maximum of five dimensions in our study).

There are two possible approaches to parallelize the evaluation of the Dean model: the first is at the subnet granularity (Figure 12a) and the second is at the clique granularity (Figure 12b). This latter approach will yield a higher level of parallelism as there are more cliques than subnet (given that a subnet can be decomposed into multiple cliques). Dependencies between the cliques may limit the number of cliques that can be evaluated in parallel at any given level within a junction tree. In this study we evaluated both approaches and found that for the networks examined, the clique based approach had a better utilization of the available processing cores. In both approaches the order in which the subnets or cliques will be evaluated is predetermined and does not vary with the network inputs.



(a) (b)

Figure 12. Parallelization of the Dean model by (a) subnets (b) cliques onto multiple cores of a processor.

3.3.2 Vectorization

As with the HTM model, there are at least two approaches to vectorization for this model: vectorizing the operations for a single image and vectorizing to evaluate multiple images simultaneously. In the former case, matrix operations would have to be vectorized as a large portion of the junction tree evaluations consist of multi-dimensional matrix operations. In the networks examined, these matrices had up to five dimensions with each dimension being up to 16 elements wide. The matrix operations included element-by-element matrix multiplies and divides. There were also matrix dimension reductions which essentially were summations along a given dimension of the matrix. Not all of these operations can be vectorized efficiently, particularly as the matrix dimensions were of small widths (that were not always multiples of the vectorization factor).

Since the model evaluates any input data in precisely the same way, multiple inputs can be evaluated in parallel through vectorization. In case of a vectorization factor of four, there will be four versions of each matrix (one for each image). The same set of operations will be carried out for all four versions of each matrix. In this case vectorization can be applied to almost 100% of all the operations.

3.4 CSRN

To perform the CSRN processing, we want to take advantage of both the task-level and data-level parallelism in the computations. As previously noted, we took advantage of the task-level parallelism within the $\mathrm{CSRN}_{\mathrm{FF}}$ and $\mathrm{CSRN}_{\mathrm{FB}}$ stages of the application and the data-level parallelism in the CCA and UWK stages. For the $\mathrm{CSRN}_{\mathrm{FF}}$ and $\mathrm{CSRN}_{\mathrm{FB}}$ stages, we map the operations of each CSRN cell element (GMLP) to a thread block. Additionally, we map the operations of each node within the GMLP to a thread within the processing block. For the CCA and UWK stage computations, we map the matrix/vector operations to a grid, such that each element within the matrix/vector operations would be processed by a thread. The matrix inversion of Γ_t is performed using a parallel Gauss-Jordan elimination technique on the GPGPU adapted from [46].

We observed that we can dramatically reduce the number of accesses to global memory in our GPGPU implementation of Gauss-Jordan elimination. Given an initial matrix (matrix to be inverted), it is appended with the Identity matrix. After an iteration of Gauss-Jordan elimination, only the shaded columns are modified. This is because of the zeros present in the remaining columns offer no change to the unobserved rows. Using this knowledge, we modified our GPGPU Gauss-Jordan elimination routine to only access global memory during the times in which there will be modification to the unobserved rows. By doing this, we were able to decrease the amount of time necessary to compute UWK computation stage by approximately 57 %.

We also observed another method to further reduce the number of global memory accesses. During the computation of the Gauss-Jordan elimination routine, the observed row is normalized by the diagonal value every iteration. This normalization actually can be postponed until after all iterations have been completed. We modified our GPGPU implementation to normalize only after all iterations of the Gauss-Jordan routine have completed. By doing this, the number of memory global accesses during UWK stage were reduced which resulted in a reduction in the computation time.

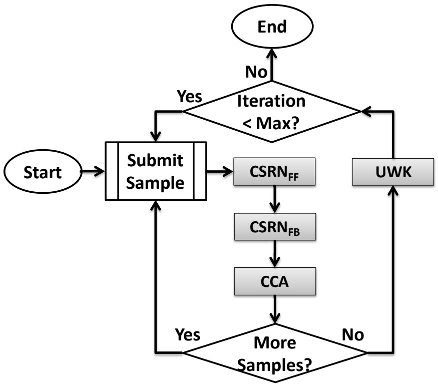


Figure 13. Flow chart for GPGPU CSRN mapping for MSEKF training. The shaded regions representation processing modules placed on GPGPU.

Figure 13 shows a representative flow chart for the GPGPU operations for training one CSRN within the network. The flow chart shown in Figure 13 would be used to train all CSRNs within the network for their respective inputs.

3.4.1 Using Multiple GPGPUs

Given the amount of inherent parallelism within the face recognition application, we wanted to more effectively take advantage of this within our processing. Therefore, we increase the amount of parallelism within our implemented CSRN based face recognition application by distributing the CSRN computation across multiple processing cores and/or GPGPUs.

For the multi-core and multi-GPGPU implementations, we use the Message Passing Interface (MPI) protocol to communicate between various multiple CPUs/GPGPUs. In this fashion, we elected to separate operations into one master process and multiple worker processes. The master process directed the operations of the worker processes while the worker processes performed the CSRN network computations. The number of CSRNs that each worker process computes varies depending upon the number of worker processes available and the number of CSRNs that are within the CSRN network. The master process will divide the work among the worker processes. The master process reads in the input and generates a work schedule for all worker processes. Then the master process uses MPI to send a personalized schedule as well as necessary inputs to each worker process. Once a worker process receives its input, it can begin operation. Once finished, the worker process will signal the master process. The master process will then retrieve the output data from the worker process. If more CSRN data needs to be processed, the master process will send the worker task a new work schedule as well as more CSRN data. The master process will continue in this fashion until all data has been processed and collected. In this fashion, both training and testing is performed.

In the multi-core implementation, a single core was used to perform the operations of a worker process. In the multi-GPGPU implementation, the worker process was performed by a single GPGPU using a core to transmit data via MPI to the master process and other worker processes. The system that we used was the Condor cluster. Access to the Condor cluster was provided by Air Force Research Laboratory (AFRL) Rome Research Site.

3.5 Implementation of RBF network on GPU Platform

3.5.1 The Cholesky decomposition algorithm

We need to invert the matrix multiplication G^TG to calculate the output/linear-weights of the RBF neural network, a relatively time consuming process. Since the resultant matrix is symmetric and positive definite, it appears that the Cholesky decomposition [47] algorithm is applicable for the inverse operation. For a given generic matrix A and its sub-matrices A_{ij}, the Cholesky decomposition algorithm can be described as follows:

$$
\begin{aligned}
A &= \begin{pmatrix} A_{11} & A^{T}{}_{21} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L^{T}{}_{11} & L^{T}{}_{21} \\ 0 & L^{T}{}_{22} \end{pmatrix} \\
&= \begin{pmatrix} L_{11}L^{T}{}_{11} & L_{11}L^{T}{}_{21} \\ L_{21}L^{T}{}_{11} & L_{21}L^{T}{}_{21} + L_{22}L^{T}{}_{22} \end{pmatrix} \qquad (31) \\
L_{21} &= A_{21}(L^{T}{}_{11})^{-1} \\
L_{22} &= A_{22} - L_{21}L^{T}{}_{21}
\end{aligned}
$$

The Cholesky decomposition algorithm can be implemented recursively and has been found to significantly reduce the computation complexity of the inversion process. In our experiment, we have employed the CUDA code developed by Volkov et al. [48] for Cholesky dcomposition.

In our experiment, we take advantage of the GPU's architecture by parallelizing some facets of the presented algorithm. The algorithm running on the GPGPU will be divided into thousands of sub-tasks and distributed to thousands of threads [49]. For calculating the distance matrix, we let each thread compute one element of the matrix. For the matrix operations (i.e., matrix multiplication for the inversion process using Cholesky's decomposition), we let each thread calculate one element of the corresponding output matrix.

Figure 14 shows the flow chart of our program. The flow chart describes a complete procedure of RBF network training on GPGPU with cross-validation (at certain spread value). The procedure can be described as follow: step 1, the program transfers the position data points from the host memory (i.e. memory of CPU platform) to the device memory (i.e. the memory of GPU platform). Step 2, to meet the requirement of cross validation algorithm, the program will divide the input position data points into two separate data sets: *"the training dataset"* and "*the testing dataset*". Step 3, after determining the *training and testing datasets*, the program will calculate the distance matrix based on the parallel algorithm mentioned above. Step 4, when the distance matrix is ready, the program will use the parallel matrix-multiplication algorithm to generate the GG^{T} matrix. Step 5, the program will continue to invert the GG^{T} matrix using Cholesky decomposition. Step 6, when the inverse operation is finished, the inverted matrix, the matrix G, and the output vector are multiplied using a parallel matrix multiplication algorithm. As discussed above, this operation will optimize the linear-weights of the RBF network. Step 7, the program will set *"testing dataset"* as the input position data points and start the validation by calculating the errors (the difference between the desired position and the RBF network's outputs). The program will repeat steps 2 to 7 based on new "training dataset" and "testing dataset" until the 7-fold cross-validation is completed. After the cross validation procedure is finished, we can calculate the average errors and set the average errors as the error of RBF network on this spread value.

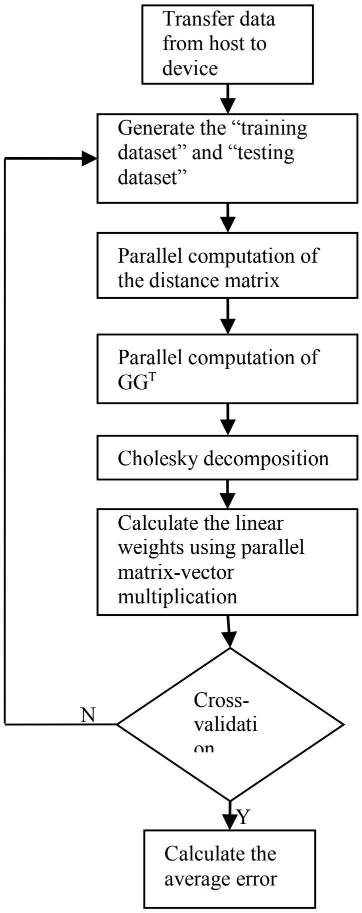

Transfer data from host to device
Generate the “training dataset” and “testing dataset”
Parallel computation of the distance matrix
Parallel computation of GG^T
Cholesky decomposition
Calculate the linear weights using parallel matrix-vector multiplication
Cross-validati on
N
Y
Calculate the average error

Figure 14. Flow chart of GPGPU implementation.

4. Acceleration results

4.1 SNN

4.1.1 Recognition of noisy images

The four networks were tested with all 48 training images and were able to recognize all the input images correctly. Figure 15 lists an additional set of test images (labeled 1 to 10) that were applied to the two networks. These include some original and partially modified versions of the first four training images (‘A’, ‘B’, ‘C’, and ‘D’). The images were applied sequentially to the inputs of all the four models. The membrane potentials of the level two neurons for the four models corresponding to these test images are shown in Figure 16. In all cases, a membrane potential of above 30 mV represented a neuron firing (and thus recognition). The Wilson and Morris-Lecar models were able to recognize all of the images except for images 5 and 6. These images were heavily modified versions of the training image ‘A’. The Izhikevich and Hodgkin-Huxley model are able to recognize all images except for image 6. The spikes produced by image 6 did not cross the 30 mV threshold needed for a proper recognition. Based on the model parameters utilized, the Izhikevich, Wilson, Morris-Lecar and the Hodgkin Huxley models required at most 14.00 ms, 3.75 ms, 0.43 ms and 1.20 ms of simulation time respectively to produce output spikes at the level two neurons. The input images were presented to the models as inputs for these simulation times for the respective models.

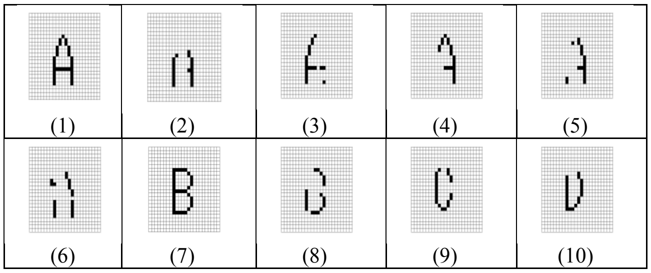
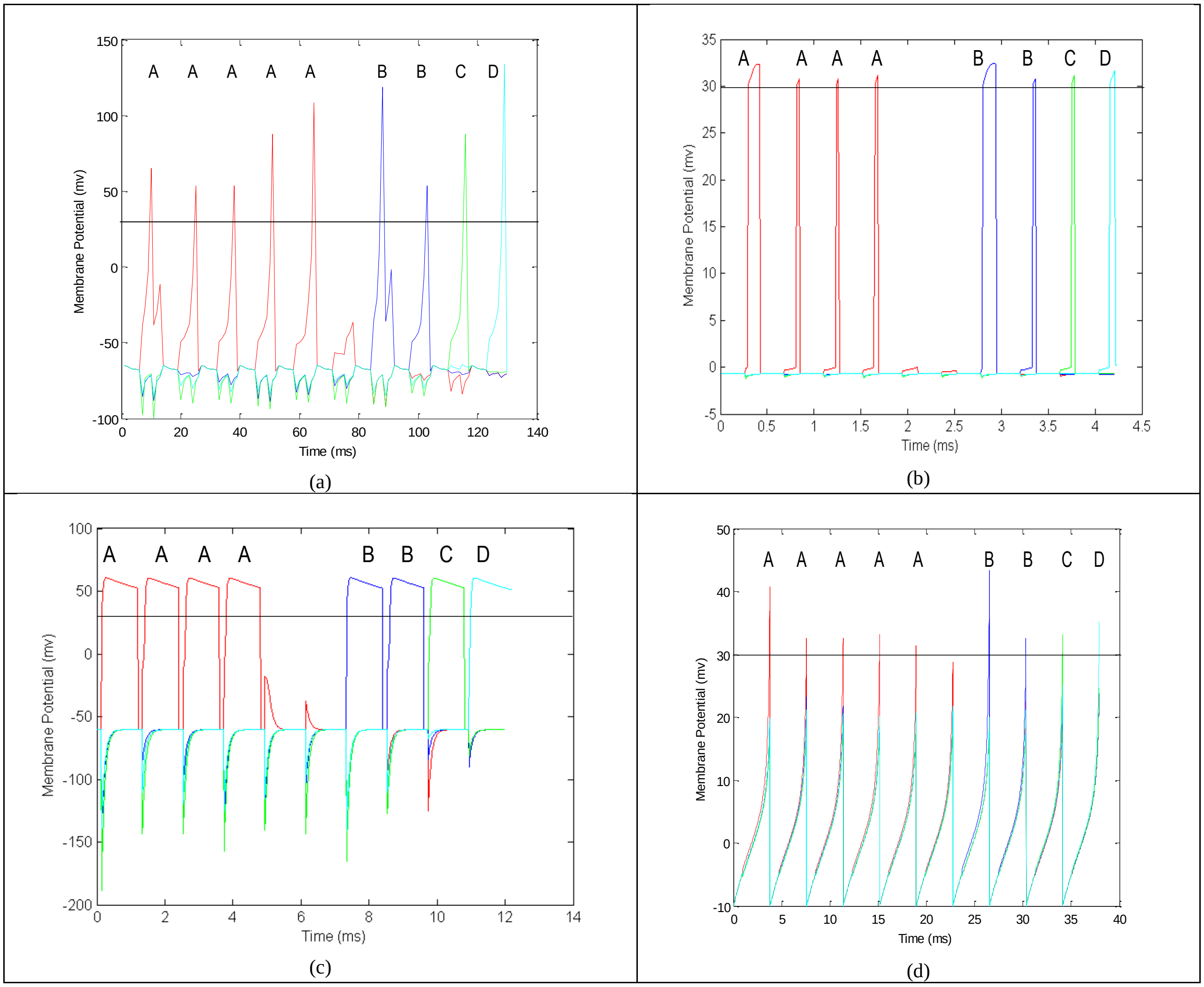


Figure 15. Additional test images.



(a)

(b)

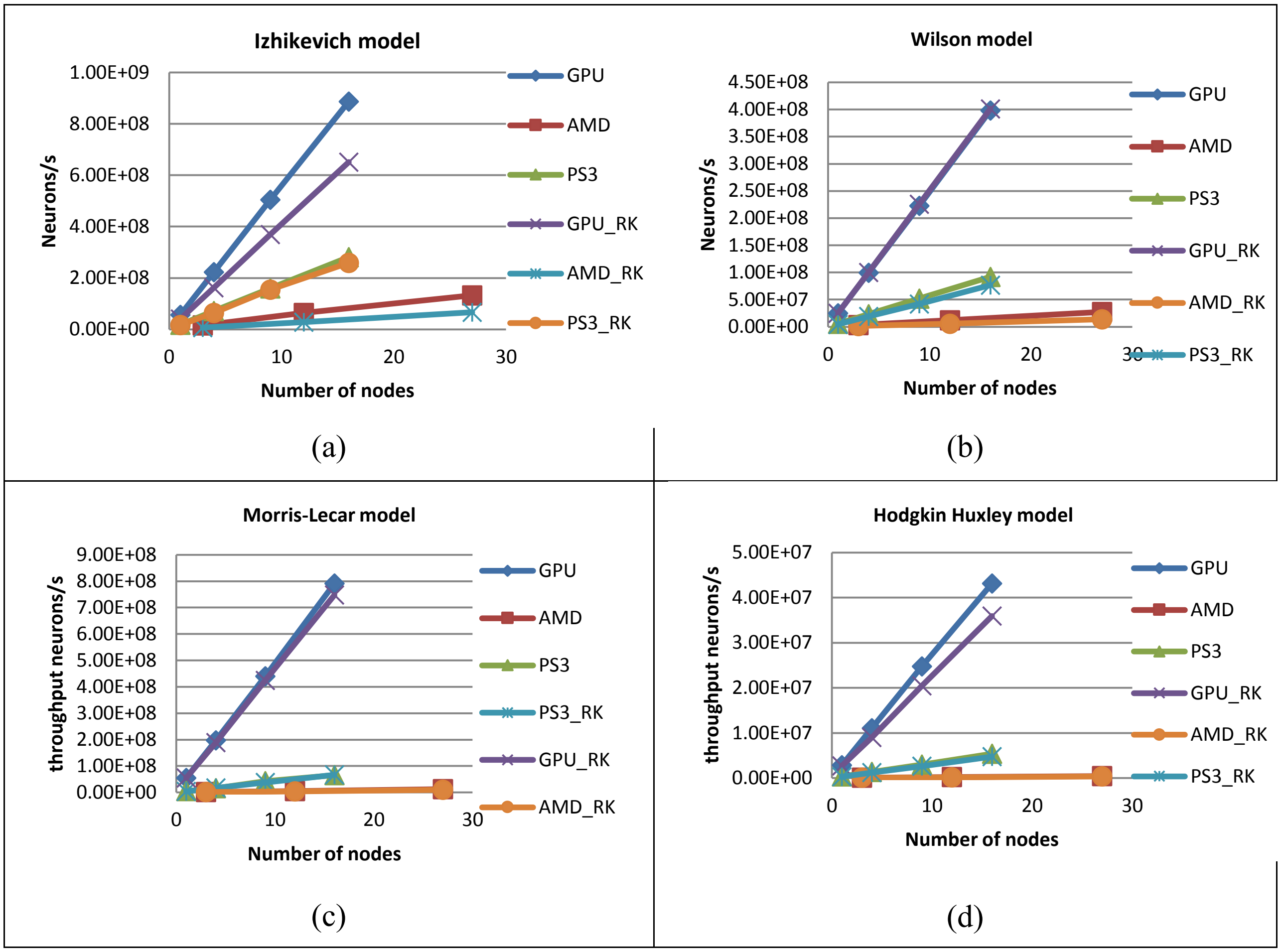
(c)

(d)

Figure 16. Level 2 neuron membrane potentials for a serial presentation of the images in Figure 8 for (a) Izhikevich, (b) Wilson, (c) Morris-Lecar, and (d) Hodgkin-Huxley models. The red line represents the membrane potential of the neuron for detecting an ‘A’, blue for ‘B’, green for ‘C’, and cyan for ‘D’.

4.1.2 Run time performance

Figure 17 shows the throughput in neurons per second of the spiking neural network models on different computing platforms for both the Euler and the Runge-Kutta methods. As expected from the flops per neuron shown in Table 1, it is seen that the Izhikevich model had the highest throughput while the Hodgkin Huxley model is slowest. In all cases, the GPU provided the highest performance. The Euler model was faster than the Runge Kutta method. Figure 18 shows the performance of the models on the AFRL PS3 cluster.



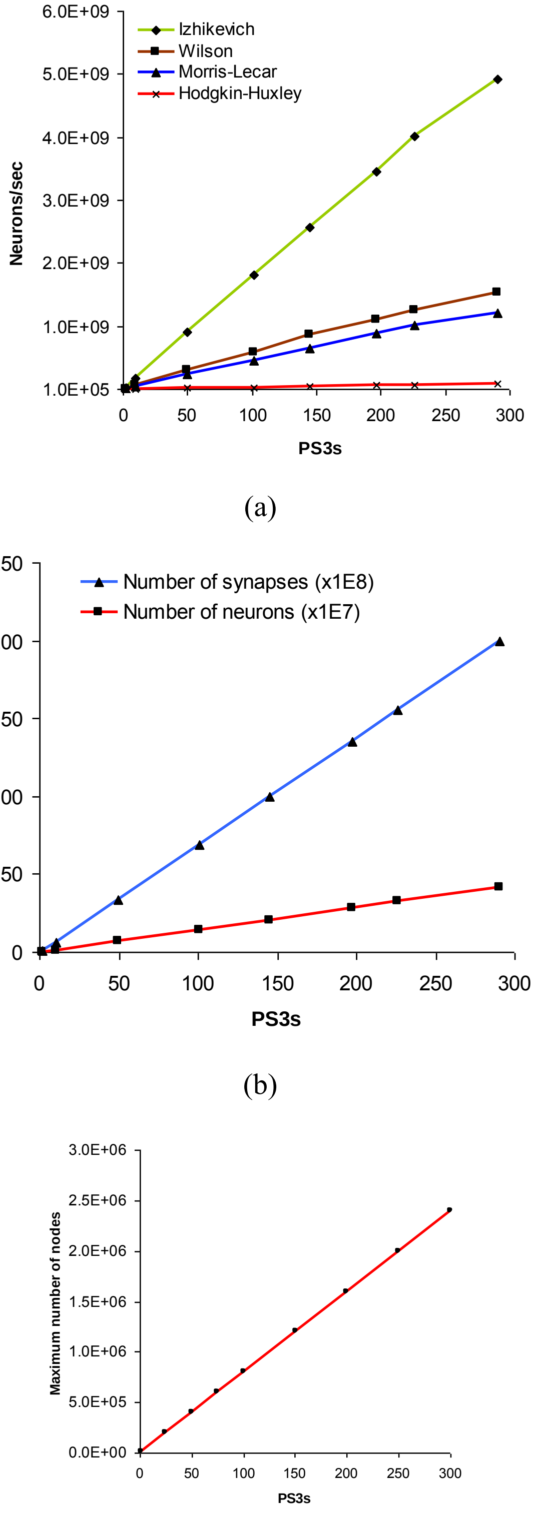
(a)

(b)

(c)

(d)

Figure 17. The Throughput of different platform at different image size and node number



(a)

(b)

(c)

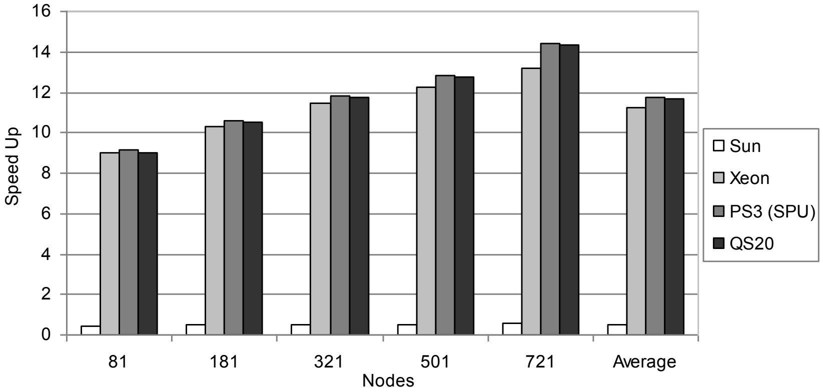
Figure 18. Scaling of the model on the AFRL PS3 cluster.

4.2 HTM and Dean models

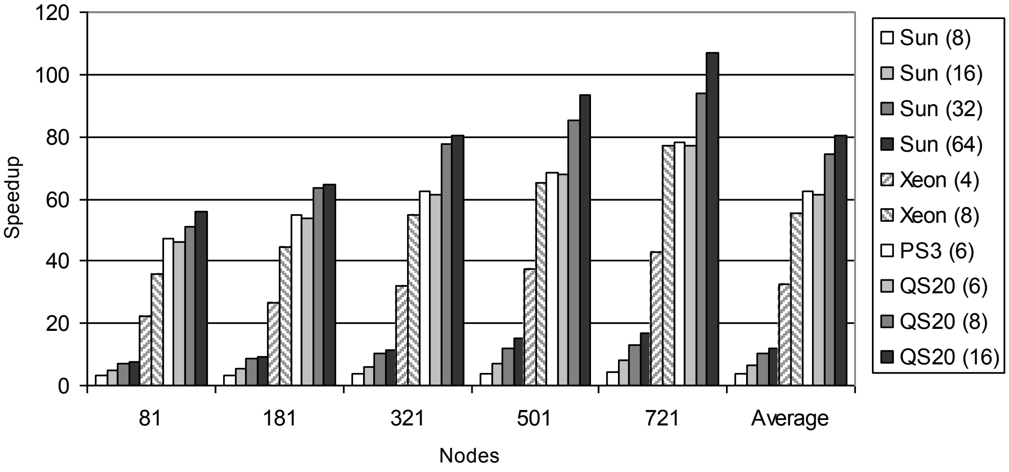
4.2.1 Speedup

Figures 19 and 20 present the performance of the HTM and Dean models respectively on the architectures examined. The speedups of the single core and multicore implementation of the models over a serial PPU implementation on the Cell processor are shown. The single core performances of the Cell SPU and Xeon are higher than the Cell PPU, while the performance of the Sun UltraSPARC T2 Plus processor was lower than on the PPU (here the UltraSPARC ran only one thread on one core). From these figures it is seen that the parallel implementations of the models provide a significant performance gain over their serial implementations. This is mainly due to the use of multiple cores and the use of vectorization on the Intel and Cell architectures. There is sufficient parallelism in the models examined, so that for all of the platforms, use of more cores provided higher speedups. Our experiments showed that increasing the number of threads on the Intel Xeon blade beyond 8 provided no further improvement in performance. The Dean model produced a higher speedup than the HTM model for all the platforms examined. It is possible that the larger number of training categories in the HTM model produces larger potential tables, which translates to more data transfers, thus limiting its speedup over the Dean model.

For both models, it is seen that the Cell processor outperformed both the Intel Xeon and the Sun UltraSPARC T2 Plus processors. The Playstation 3 with 6 available SPU cores outperforms the Intel Xeon processor (with 4 cores) by about 1.9 times for the HTM model and by 2.4 times for the Dean model. As a result the Playstation 3 also outperformed the blade with two Intel Xeon processors. The speedup of the Cell processor on the QS20 with all 8 SPU cores available over a single Intel Xeon processor was about 2.3 times for the HTM model and about 3 times for the Dean model. Utilizing both Cell processors on the QS20 (16 threads) provides only a 11% performance gain for HTM model and a 22% performance gain for the Dean model over one Cell processor (8 threads). We believe this is due to the memory accesses becoming a bottleneck as calculation times become close to data access times (as shown in Table 6). This effect is not seen on the Sun processor when going from 8 to 16 threads as the calculations take much longer on that system.

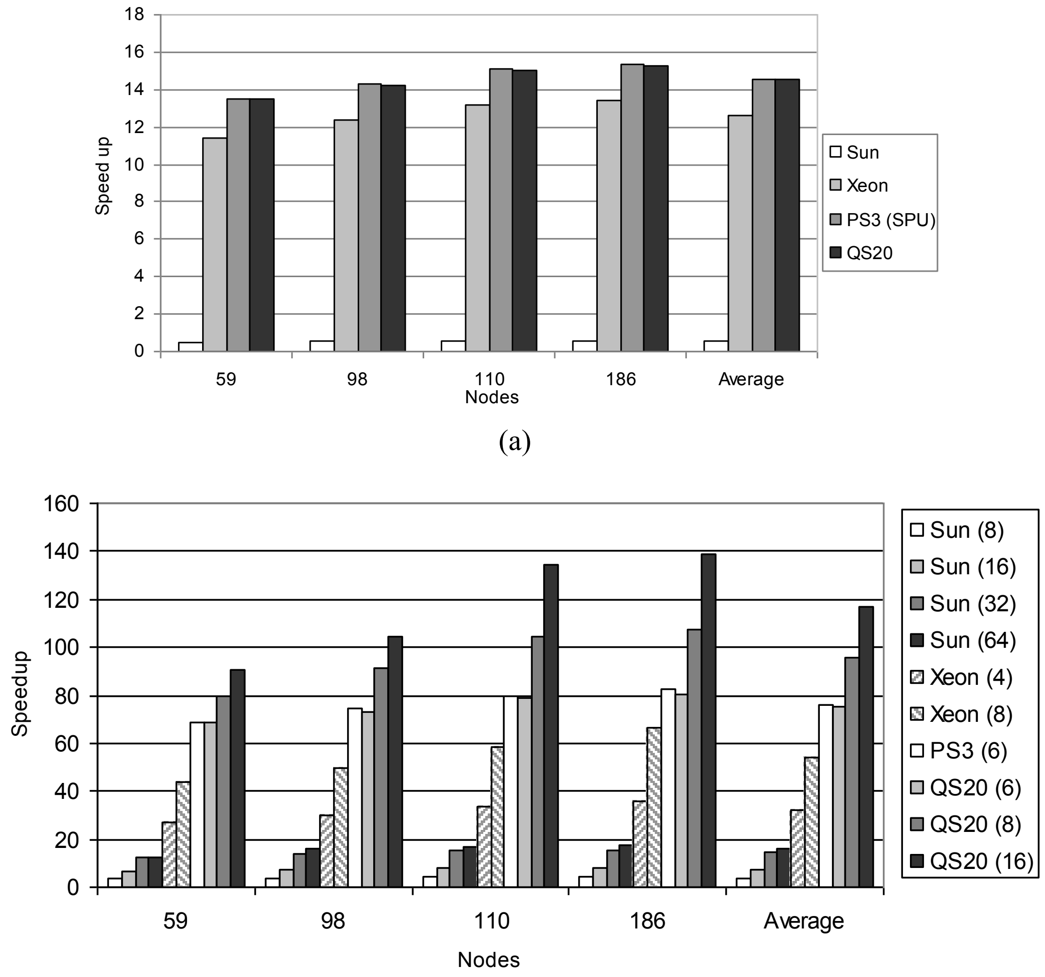


(a)



(b)

Figure 19. Speedup over the Cell PPU for the HTM model on different multicore architectures. Part (a) shows the speed ups for single thread implementations. Part (b) shows the speed ups for multi-thread implementations. The numbers in parenthesis in the legend represent the number of threads utilized on each platform.



(a)

(b)

Figure 20. Speedup over the Cell PPU for the Dean model on different multicore architectures. Part (a) shows the speed ups for single thread implementations. Part (b) shows the speed ups for multi-thread implementations. The numbers in parenthesis in the legend represent the number of threads utilized on each platform.

On the UltraSPARC processor, the Dean model provides speedups of about 2 when going from 8 to 16 threads and when going from 16 to 32 threads. The speedup from 32 to 64 threads is minor (about 1.1 times for the 186 node network). The HTM model provided lower speedups than the Dean model: 1.9 times for the largest model tested when going from 8 to 16 threads, 1.7 times when going from 16 to 32 threads, and 1.3 times when going from 32 to 64 threads.

The Sun processor provides a lower speedup than the Xeon and Cell processors because of a lower clock frequency and a lack of vector capabilities. If multiple images were not available to process simultaneously (such as if there were only one small camera source), then we would not be able to take advantage of the vectorization utilized. In this case the performance of the Intel and Cell architectures would be about one fourth of their current values. Since the Sun does not support vector operations, its performance would not be affected. In this situation, the Sun processor with 64 threads would actually be faster than the Xeon processor with 4 threads; about 2 times for the largest HTM model and 1.6 times for the largest Dean model.

4.2.2 Runtime breakdown of models

Figures 21 and 22 show the runtime breakdowns of the HTM and Dean models respectively on the Cell processor (on the Playstation 3) and the Intel Xeon processor (4 thread implementation). The runtime break downs are given for the smallest and the largest network sizes for both the models. This is done to compare the change in each part of the algorithm with the scaling of the model. The time for signaling between the different threads on all the platforms was insignificant due to the pre-assigning of nodes to different threads at the start of the program. Therefore this time is not listed separately in the timing breakdown.

For the Cell platform, the non-overlapped memory access time is calculated by taking the difference between the overall runtime of the application and the runtime with DMA data transfers commented out of the code. This is the part of the DMA accesses that could not be overlapped with computations (generally through double buffering). On the Intel Xeon platform, this time was calculated by taking the difference between the overall runtime and a version of the code with all global variables in the threads converted to local variables (synchronization barriers between threads were kept intact). The number of computations (array accesses and other operations) was kept the same in both cases.

The results show that on the Cell processor, DMA transfers that could not be overlapped can be a significant percentage of the overall runtime. However this fraction decreases as the network sizes increase since the nodes in the network become more complex and thus have more computations to be carried out per node. This is seen by the increase in the computation percentage for equations 2, 3, and 4 in the HTM model and in the percentage of time for getting evidence in the Dean model. Stalls due to global variable accesses on the Xeon processor (listed as non-overlapped memory access in Figures 21 and 22) showed similar trends as well.

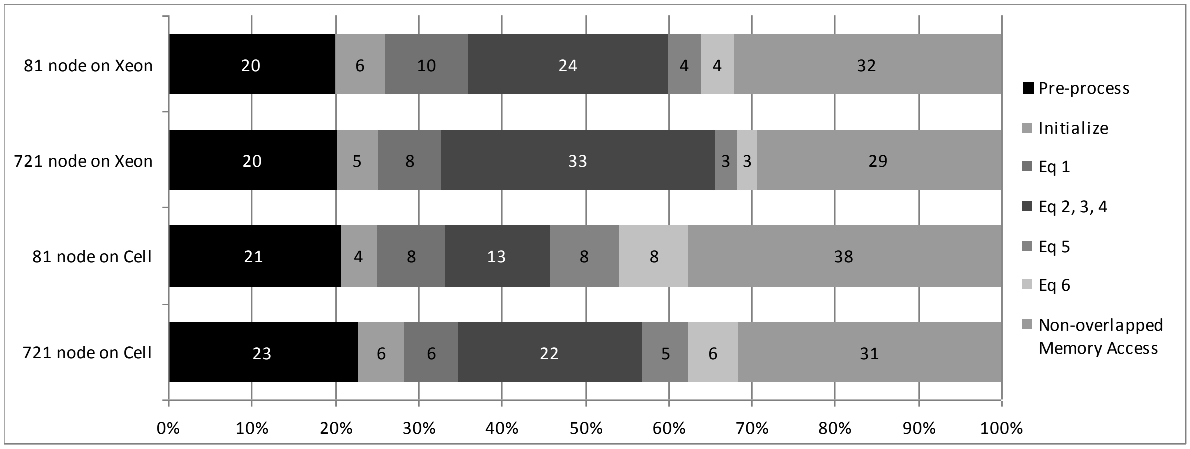


Figure 21. Runtime breakdowns for the HTM model on the PS3 and Xeon processors (a) Runtime breakdown for the 81 node network on the Playstation 3. (b) Runtime breakdown for the 721 node network on the Playstation 3. (c) Runtime breakdown for the 81 node network on the Xeon Processor. (d) Runtime breakdown for the 721 node network on the Xeon Processor.

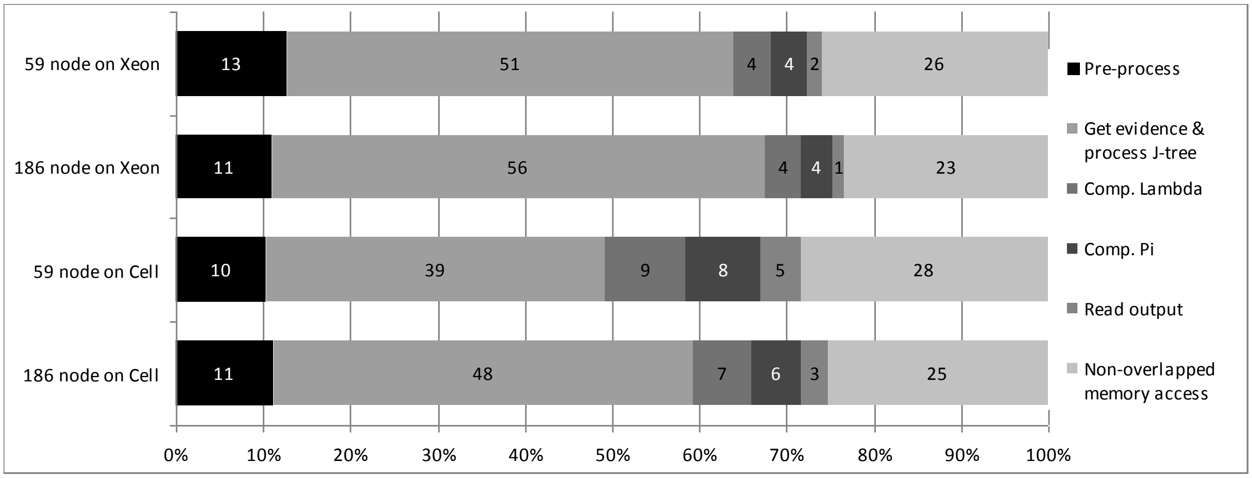


Figure 22. Runtime breakdowns for the Dean model on the PS3 and Xeon processors (a) Runtime breakdown for the 59 node network on the Playstation 3. (b) Runtime breakdown for the 186 node network on the Playstation 3. (c) Runtime breakdown for the 59 node network on the Xeon Processor. (d) Runtime breakdown for the 186 node network on the Xeon Processor.

The overall DMA is unlikely to change with the number of cores used on the Cell processor as these accesses go to a centralized memory system. However the overall computation time is likely to decrease with more cores due to increased parallelism. Hence, as the number of cores increase, the DMA time can exceed the computation time, thus limiting the speedup seen with increasing cores. This effect is seen in Figures 19 and 20: the speedup does not double when going from 8 to 16 cores. Although this could be due to the impact of off-chip memory buses, the results in Table 6 seem to indicate that it is due to a memory bottleneck. Table 6 shows the runtime breakdown of the largest HTM and Dean model networks on the Cell processor platforms examined: Playstation 3 with 6 SPU, and QS20 with both 8 and 16 SPUs. While the computation time decreased with increasing numbers of cores, the non-overlapped DMA time increased slightly (since the computations started taking less time than data transfers). To alleviate this issue, a higher memory bandwidth would be needed. This would be seen by having each cell processor have access to its own dedicated memory.

Table 6. Run time break down of the largest HTM and Dean models on the QS20 with 6, 8, and 16 threads. All times are in ms.

	HTM			Dean		
SPUs	6	8	16	6	8	16
Computation only (ms)	7.51	5.45	3.50	12.10	8.10	4.50
Non-overlapped DMA (ms)	3.45	3.60	4.45	4.10	4.34	5.12
Total (ms)	10.96	9.05	7.95	16.20	12.44	9.62
% of DMA in runtime	31.47	39.77	55.97	25.30	34.88	53.22

4.2.3 Parallelization strategy for the Dean model

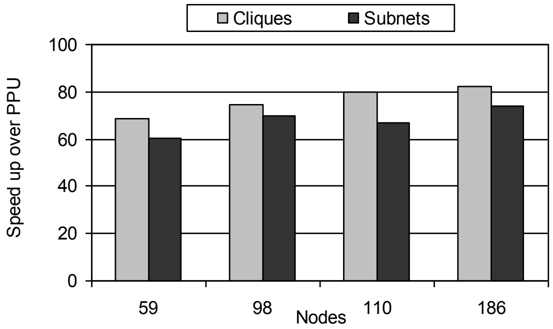
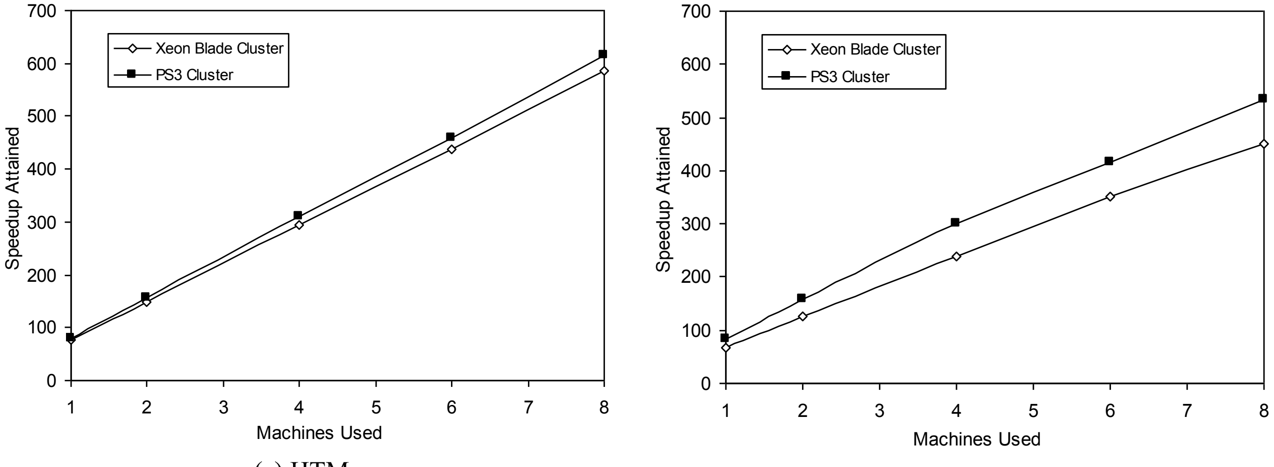


Figure 23. Parallelization of the Dean model by cliques vs. subnets. The 59 and 110 node networks are using only 4 SPUs because of the limited set of subnets on those networks. The other two are using six SPUs.

Figure 23 compares the two parallelization approaches examined for the Dean model: clique based and subnet based. All the subnets in a layer can be evaluated in parallel. The networks with 59 and 110 nodes had fewer subnets in level 1 than the 98 and 186 node networks. Thus the former set of networks provided lower speedups than the latter set when parallelized by subnets. For all the networks, there were more cliques that could be evaluated in parallel than subnets (since each subnet could be decomposed into multiple cliques). Thus the clique based parallelization approach provided higher speedups for all the network sizes evaluated.

4.2.4 MPI Implementations

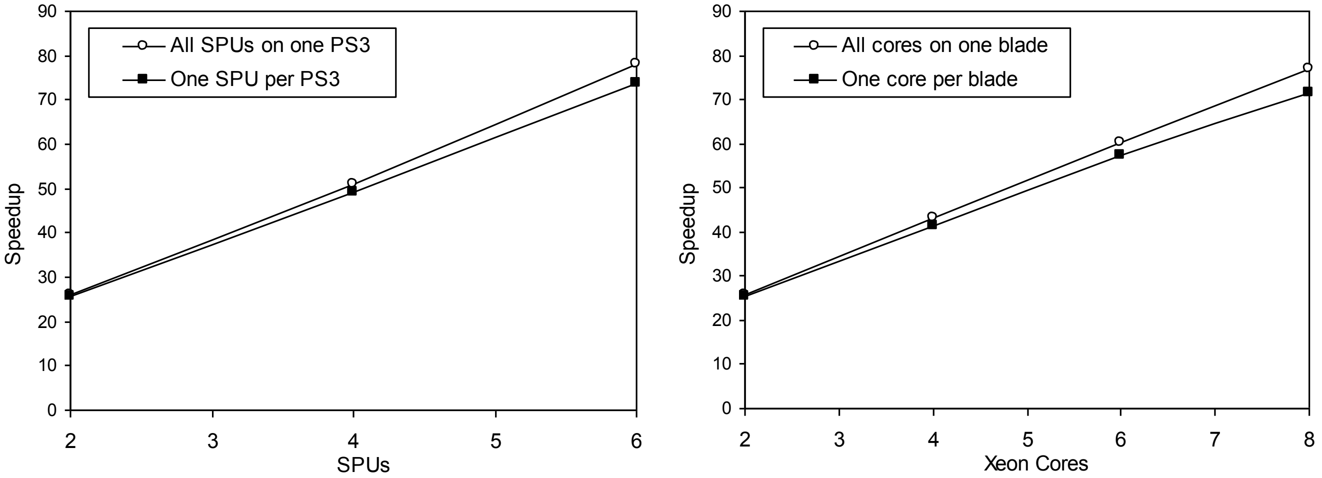


(a) HTM

(b) Dean

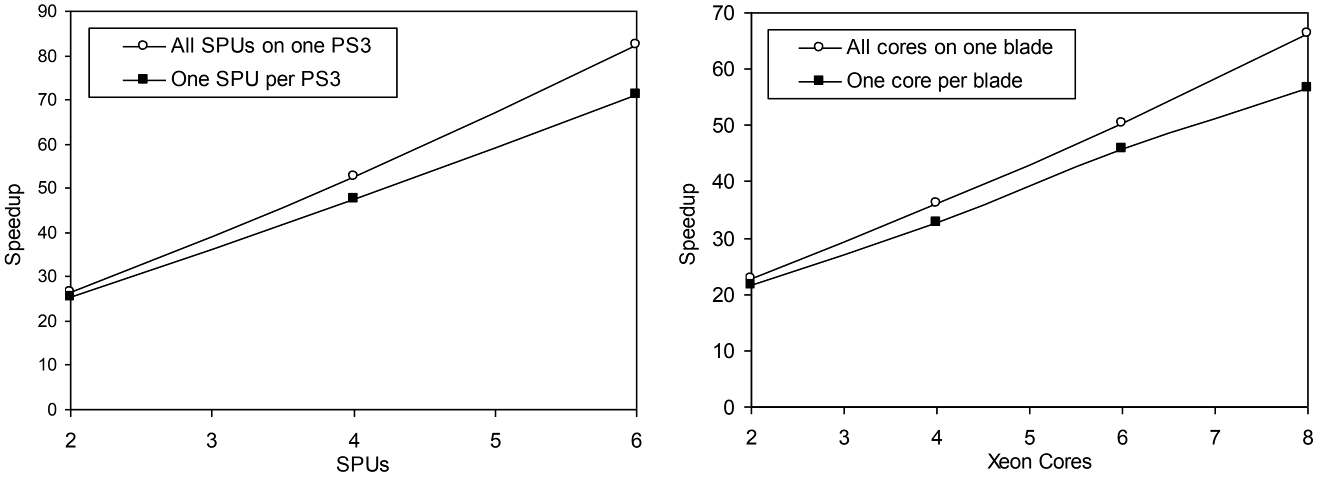
Figure 24. Performance scaling of the MPI based implementations with increasing number of machines used. The largest version of the models examined.

Figure 24 shows the performance scaling of the MPI implementation of each model. Two clusters consisting of PS3s and Xeon blades were utilized with all the cores on each machine being used. The largest network for each model was implemented (721 nodes for the HTM model and 186 nodes for the Dean model). The performance of both models scaled with the increase in the number of cores used. The Dean model however has a lower performance gain than the HTM model. This is because 1) the HTM model has more computation units (576 nodes in layer 1 of HTM vs. 225 cliques in layer 1 of Dean), and 2) the Dean model has higher connectivity between its cliques than the HTM model has between its nodes. This leads to lower parallelism and higher communication in the Dean model compared to the HTM model.



(a) (b)

Figure 25. Comparison of multithreading versus MPI implementations of the HTM model on the (a) ARSC PS3 cluster, and (b) Palmetto Xeon cluster. The number of cores were varied with all core being either on one machine (multithreaded implementation) or one core per machine (MPI implementation).



(a) (b)

Figure 26. Comparison of multithreading versus MPI implementations of the Dean model on the (a) ARSC PS3 cluster, and (b) Palmetto Xeon cluster. The number of cores were varied with all core being either on one machine (multithreaded implementation) or one core per machine (MPI implementation).

Figures 25 and 26 examine the impact of a purely MPI based parallelization scheme over a purely multicore parallelization approach. The MPI based parallelization scheme used only one compute node per machine, with the machines communicating through MPI. The multicore approach utilized only the cores available on a single machine. The results show that in all cases, the MPI based approach had a lower performance than a multicore based approach. This is primarily due to the higher communication cost of MPI (communications have to go through multiple network interface cards and the network connecting the different machines).

In the MPI implementation of a model, the data needed for computation on a machine can come either from the machine's local memory (a local transfer) or from another machine (an MPI transfer). As the number machines is increased, the total amount of data needed for a model will not change, but the fraction of data coming over MPI could vary. This effect is shown in Table 7, where the models are run on varying numbers of PS3s, with only one core utilized per PS3. It is seen that fraction of MPI transfers increases for the Dean model, but not for the HTM model. This is because the nodes in the HTM model are not as densely connected as the cliques in the Dean model. In fact, on the HTM model, the network can be split at a single point (the root layer), thus making the fraction of MPI transfers constant. As a result of this trend, the drop in performance of the MPI approach over the multicore approach was higher for the Dean model.

Table 7. Distribution of data transfers from local memory and between PS3s (MPI transfers) for the: (a) HTM model, and (b) Dean model with variation in number of machines used.

PS3s	**1**	**2**	**4**	**6**	**8**
Local transfers	3605	3317	3317	3317	3317
MPI transfers	0	288	288	288	288
Total transfers	3605	3605	3605	3605	3605
% MPI of total	0	7.99	7.99	7.99	7.99

(a)

PS3s	**1**	**2**	**4**	**6**	**8**
Local transfers	1112	976	932	902	888
MPI transfers	0	136	180	210	224
Total transfers	1112	1112	1112	1112	1112
% MPI of total	0	12.23	16.19	18.88	20.14

(b)

4.3 CSRN Acceleration results

4.3.1 Maze traversal

4.3.1.1 Single-core CPU versus GPGPU

Two initial designs were implemented: an optimized CPU version using the C programming language and OpenCV library [50,51] and a GPGPU version using both the C programming language and CUDA [52]. The CPU implementation was performed on a 2.67 GHz Intel Xeon X5550 processor, and the GPGPU implementation was performed using a combination of the same Intel Xeon processor and an NVIDIA Tesla C2050 GPGPU. The GPGPU performed the CSRN operations while the Intel Xeon processor was used for setup and communication purposes. The CPU implementations were compiled using GCC version 4.1.2 (GCC).

Figure 27 shows the timing results of both implementations being used for the CSRN maze traversal training phase. In this set of tests, the number of rows and columns for the maze input samples were varied as shown along the x-axis. For each input maze size, five samples were used. The total runtime to perform the training is shown along the y-axis. All other parameters remained fixed. As can be seen in Figure 27, the GPGPU implementation is slightly faster than the CPU implementation. For this set of tests, the best observed speedup was 3.04 times with an average speedup of 2.13 in favor of the GPGPU implementation.

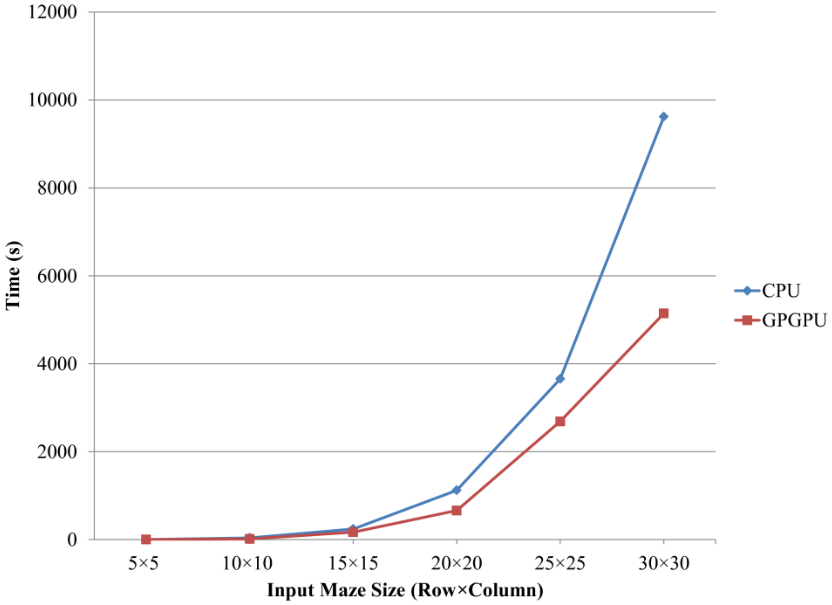


Figure 27. Maze traversal CPU vs GPGPU training runtimes.

Figure 28 shows the timing results of both implementations being used in the CSRN maze traversal testing phase. As in the set of tests for training, the number of row and columns for the maze input samples were varied as shown along the x-axis, while all other parameters remained constant. In addition, 15 samples were used for each input maze size. The total runtime to perform the testing is shown along the y-axis. Akin to the results shown in Figure 27, the GPGPU implementation is faster than the CPU implementation. For this series of tests, the best observed speedup was 4.39 times with an average speedup of 3.55 in favor of the GPGPU implementation.

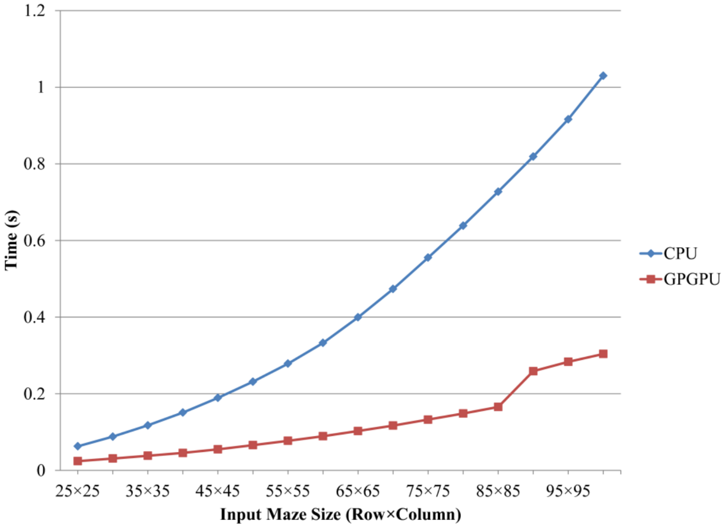


Figure 28. Maze traversal CPU vs GPGPU testing runtimes.

4.3.1.2 *GPGPU design extensions*

To achieve higher speedups, several design extensions were examined. Specifically, the speedup performance of the training phase was closely examined as it is the limiting factor when processing larger datasets. For example, to train a CSRN using five 30×30 maze input samples required 5,143.12s while testing using five 30×30 maze input samples required 13.82ms on the GPGPU. Therefore, the average time required to compute each stage was measured to capture the most time consuming part of the training phase. For these measurements, the rows and columns of the input samples were varied (5×5 to 15×15 in 5×5 increments), and the number of input samples was fixed at five. Table 8 shows these results of our measurements.

Table 8. Timing breakdown for computation stages of GPGPU design

Computation Stage	**Input Maze Size (Row×Column)**		
	5×5	**10×10**	**15×15**
$CSRN_{FF}$	2.07%	0.43%	0.06%
$CSRN_{FB}$	3.57%	0.62%	0.08%
CCA	1.04%	0.17%	0.02%
UWK	**76.96%**	**97.11%**	**99.68%**
Overhead	16.36%	1.67%	0.17%

As highlighted in bold in Table 8, the limiting factor is clearly the *UWK* stage as it takes up the bulk of the computation. This result is not surprising as the *UWK* stage includes a time consuming matrix inversion within its computations. In order to achieve improved speedup performance, the *UWK* stage must be examined carefully for possible modification. Therefore, the initial GPGPU implementation was modified following three extensions.

1) *Data caching of specific variables*:

   One method of decreasing the computation time of the *UWK* stage is to incorporate caching. The slowest part of many GPGPU applications is the time required to access global memory. By caching the data acquired from global memory accesses, one can save time. From Equations (**Error! Reference source not found.**) – (**Error! Reference source not found.**) of the *UWK* stage, the variables C_t, G_t, α_t, and K_t can be cached because of their repeated use. The initial GPGPU implementation was modified to incorporate caching for the aforementioned variables in the *UWK* stage. Unfortunately, the *UWK* stage showed insignificant improvement from taking this action. This is mainly a result of the additional computation required to processing 64-bit data (which we required to maintain precision). Caching works better for data memory elements of 32 bit-width or less.

2) *Reducing global memory accesses*:

   Since adding data caching showed little effect, another method of decreasing the computation time of the *UWK* stage was tried. As can be observed in the GPGPU implementation of Gauss-Jordan elimination, one can dramatically reduce the number of accesses to global memory. After an iteration of Gauss-Jordan elimination, only the shaded regions are modified. This is because of the zeros present in the remaining areas offer no change to the non-pivot rows. Using this knowledge, the GPGPU Gauss-Jordan elimination routine was modified to only access global memory during the times in which there will be modification to the non-pivot rows. By doing this, the amount of time necessary to compute *UWK* computation stage decreased by approximately 57% when compared to the initial implementation.

3) *Saving Gauss-Jordan normalization until end of computation*:

During the computation of the Gauss-Jordan elimination routine, the pivot row is normalized by the diagonal value every iteration. This normalization actually can be postponed until after all iterations have been completed. Therefore, the GPGPU extension 2 implementation was updated to normalize only after all iterations of the Gauss-Jordan routine were completed. By doing this, the number of global memory accesses during the *UWK* stage were reduced which resulted in a slight reduction in the computation when compared to extension 2 alone.

Figure 29 shows the effect that each extension had on the overall time when compared against the CPU implementation. The most significant overall speedup occurs as a result of extension 3. The best observed speedup acquired from extension 3 is 8.32 times with an average speedup of 7.28 times.

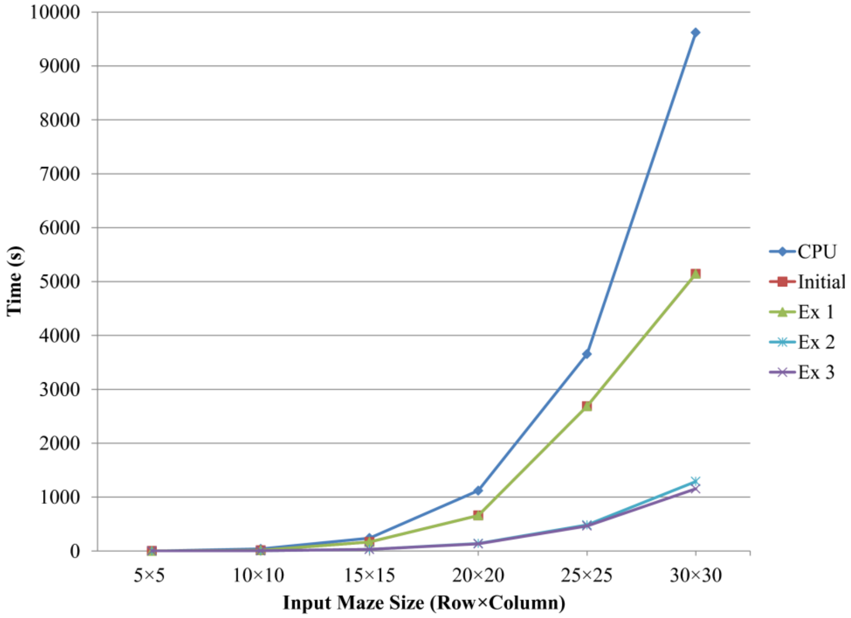


Figure 29. CPU vs GPGPU with extensions training runtimes. Initial refers to our original GPGPU implementation. Ex 1, 2, and 3 refer to our incorporation of extensions 1, 2, and 3 into our original GPGPU design.

4.3.2 Face recognition

4.3.2.1 Improving the GPGPU design

For developing the face recognition GPGPU CSRN implementation, the extension 3 maze traversal GPGPU CSRN design was used as a base. In that design, only one CSRN cell computation was performed per thread block. Given the small size of the maze traversal CSRN cell GMLP (17 nodes), many threads were left idle. To improve the GPGPU utilization, the number of CSRN cell computations performed per thread block was increased.

Since the face recognition CSRN cell GMLP is a 27 node network, the operations of one GMLP can be performed using the threads associated with one warp (32 threads). Given that a thread block can process multiple warps, one thread block can be utilized to process several different CSRN cells simultaneously exploiting instruction-level parallelism. For the face recognition GPGPU CSRN design, four different CSRN cells were processed per thread block as this was observed to achieve optimal design performance. During the $\mathrm{CSRN}_{\mathrm{FF}}$ and $\mathrm{CSRN}_{\mathrm{FB}}$ stages, shared memory is used to store repeatedly used data such as *W* and *ww* during computation. This reduces the number of times that global memory is accessed by copying the data down into shared memory at the start of the process. The use of multiple warps benefits from this because the shared *W* and *ww* data can be used among the various CSRN cells being computed within a thread block thus reducing the amount of traffic seen by global memory. Using these CSRN cell computation advancements in addition to using better GPGPU memory management, the CSRN GPGPU design improved.

4.3.2.2 Single-core CPU verses GPGPU

Two initial designs were implemented to examine how well CSRN based face recognition maps to a GPGPU system. Akin to the maze traversal designs, the two initial face recognition designs were an optimized CPU version using the C programming language and OpenCV library and a GPGPU version using both the C programming language and CUDA. The CPU implementation was performed on a 2.67 GHz Intel Xeon X5650 multi-core processor and the GPGPU implementation was performed on a combination of the same Intel Xeon multi-core processor and the NVIDIA Tesla C1060, C2050, and C2070 GPGPUs.

One GPGPU design was developed for use among the three GPGPU platforms. One expects the performance between the C2050 and C2070 GPGPUs to be identical because the architectures are the same with the exception of global memory. The C2070 GPGPU has twice the amount of global memory. However, using the newer generation GPGPUs (C2050s and C2070s) can be expected to result in better performance than the older generation GPGPUs (C1060s). A contributing reason is because the C2050 and C2070 have more SPs (the C2070 and C2050 have 208 more SPs than the C1060). In addition, the C2050 and C2070 have the ability to perform up to four concurrent streams of independent processing, while the C1060 performs one stream. Within the GPGPU design, large numbers of SPs as well as multiple concurrent streams are utilized whenever possible. All three GPGPU implementations will offer improvements over an equivalent CPU implementation; a significant edge will go to the C2050 and C2070.

During the experiments, the CPU implementation was tested using both the GNU compiler 4.1.2 (GCC) and Intel compiler version 12.0.0 (ICC). The Intel compiler further optimizes code to take advantage of SIMD (single instruction multiple data) instructions. Lastly, the face sequences for the experiments were taken from the Sheffield face database [53].

For using CSRN based face recognition, only the CSRN training phase of the algorithm is accelerated since this portion is the most time consuming. In doing so, four experiments were conducted exercising how changes in the CSRN network parameters affect GPGPU acceleration. The CSRN network parameters observed were number of people classes, PCA components, samples per face sequence, and number of face sequences.

Experiment 1: Varying people classes

The first experiment involved testing the single-core CPU and single GPGPU implementations using an increasing number of people classes. The number of classes varied from one to 10. Each people class consisted of a group of 10 CSRNs (10 PCA components) using face sequences that have been sampled nine times. Only one face sequence per people class was used during training. The training was performed using MSEKF. The training times for the experiment are shown in Table 9.

Table 9. Training times for increasing number of people classes. The implementations shown are GCC single-core CPU (G-CPU), ICC single-core CPU (I-CPU), C1060 GPGPU, C2050 GPGPU, and C2070 GPGPU.

People Classes	G-CPU (s)	I-CPU (s)	C1060 GPGPU (s)	C2050/C2070 GPGPU (s)
1	189.67	33.18	11.12	6.48
2	379.27	66.24	21.06	12.29
3	568.84	99.32	31.09	18.44
4	757.71	132.38	41.47	24.66
5	948.04	165.45	52.01	30.82
6	1,137.66	198.53	62.09	38.92
7	1,327.19	231.62	72.87	42.98
8	1,516.62	264.72	82.61	51.78
9	1,706.18	297.79	92.96	55.44
10	1,891.93	330.83	103.8	61.64

From the training times shown in Table 9, the average speedup for the ICC single-core CPU, C1060 GPGPU, and C2050/C2070 GPGPU versions over the GCC single-core CPU compilation is 5.73, 18.13, and 30.34 times respectively. Furthermore, the C1060 GPGPU and the C2050/C2070 GPGPU carry an average speedup over the ICC single-core CPU of 3.17 and 5.30 times respectively. Given that the C2050/C2070 GPGPU implementations are faster than the C1060 implementation over all scenarios, the inclusion of the C1060 GPGPUs will play a limiting factor in multi-GPGPU experiments. As the number of people classes increased, the performance benefit achieved from using GPGPUs stayed mostly the same. This can be expected since increasing the number of classes is equivalent to adding more CSRNs to the network.

Experiment 2: Varying PCA components

The second experiment varied the number of PCA components. In varying the number of PCA components, the total number of CSRNs used to represent a person changed. Therefore, varying the number of PCA components should have the same effect as varying the number of classes. For this experiment, the number of PCA components varied from 10 to 25 in increments of five for five people classes. As in the first experiment, one face sequence sampled nine times per people class was used. The training times for this experiment are shown in Table 10.

Table 10. Training times for increasing the number of PCA components. Implementations shown are for GCC single-core CPU (G-CPU), ICC single-core CPU (I-CPU), C1060 GPGPU, C2050 GPGPU, and C2070 GPGPU.

PCA Components	G-CPU (s)	I-CPU (s)	C1060 GPGPU (s)	C2050/C2070 GPGPU (s)
10	948.08	165.45	51.71	30.77
15	1,421.94	248.16	77.53	48.54
20	1,895.89	330.84	103.48	61.45
25	2,369.68	413.55	129.28	76.94

From the results shown in Table 10, the average speedup performances are nearly identical to the average speedups observed in the previous experiment. The average speedup for the ICC single-core CPU, C1060 GPGPU, and C2050/C2070 GPGPU versions are 5.73, 18.33, and 30.44 times respectively over the GCC single-core CPU. Likewise, the C1060 GPGPU and C2050/C2070 GPGPU versions are 3.20 and 5.31 times greater than the ICC single-core CPU compilation.

Experiment 3: Varying samples per face sequence

The third experiment observed how the number of samples per face sequence changed the runtime performance of the application. As the number of samples per face sequence increases, the number of cells per CSRN increases. Also, the size of Γ_t (from Equation (**Error! Reference source not found.**)) is a square of the number of samples per face sequence. Therefore, increasing the number of samples per face sequence will increase the time needed to perform the required matrix inversion. To evaluate this, a third experiment was performed where the number of samples per face sequence varied (25, 36, 49, 81, 100, 121 and 144). There are not enough samples per face sequence in the Sheffield face database to support sampling the sequence 64, 81, 100, 121, and 144 times; therefore, randomly generated data was used for those data points across different classes. In this experiment, 25 PCA components were used. Table 11 shows the runtime results of this experiment as well as the increasing Γ_t size. Table 12 shows the resultant speedups associated with the runtime results shown in Table 11.

Table 11. Training times for increasing number of samples per face sequence. The implementations shown are GCC single-core CPU (G-CPU), ICC single-core CPU (I-CPU), C1060 GPGPU, C2050 GPGPU, and C2070 GPGPU. In addition, the row×column size of Γ_t is shown as the number of sample per face sequence increases.

Samples Per Face Sequence	**Γ_t Size**	**G-CPU (s)**	**I-CPU (s)**	**C1060 GPGPU (s)**	**C2050/C2070 GPGPU (s)**
25	25×25	2,366.80	413.55	133.63	75.82
36	36×36	2,658.56	470.09	132.01	79.99
49	49×49	3,024.64	538.06	158.36	90.55
64	64×64	3,471.21	622.97	184.54	92.54
81	81×81	3,999.29	730.21	211.14	105.38
100	100×100	4,667.80	861.59	237.94	119.02
121	121×121	5,480.38	1,022.87	298.65	136.31
144	144×144	6,465.59	1,209.16	322.11	145.92

Table 12. Speedup comparison using the runtimes shown in Table 11. The implementations shown are GCC single-core CPU (G-CPU), ICC single-core CPU (I-CPU), C1060 GPGPU, C2050 GPGPU, and C2070 GPGPU.

Samples Per Face Sequence	**Speedup over G-CPU**			**Speedup over I-CPU**	
	I-CPU	**C1060 GPGPU**	**C2050/C2070 GPGPU**	**C1060 GPGPU**	**C2050/C2070 GPGPU**
25	5.72	17.71	31.22	3.09	5.45
36	5.66	20.14	33.24	3.56	5.88
49	5.62	19.1	33.4	3.4	5.94
64	5.57	18.81	37.51	3.38	6.73
81	5.48	18.94	37.95	3.46	6.93
100	5.42	19.62	39.22	3.62	7.24
121	5.36	18.35	40.21	3.42	7.5
144	5.35	20.07	44.31	3.75	8.29

Here, the trend is that the performance benefit offered by the GPGPU implementation improves as the face sequence sampling increases. There is a moderate performance benefit for the C1060 GPGPU implementation. However, there is a significant performance benefit for the C2050/C2070 GPGPU implementation. The speedup performance shown in the GPGPU implementations is expected because the number of CSRN cells computations increase one-to-one with the number of samples per face sequence. For the single-core CPU implementations, these additional CSRN cell computations are performed serially whereas they are performed in parallel for the GPGPU implementations.

Furthermore, the inversion of Γ_t for the GPGPU implementations can be performed much faster than the single-core CPU implementations. This is a direct result of the parallelism involved in the GPGPU matrix inversion scheme. The larger the size of Γ_t, the greater the advantage will be in favor of the GPGPU implementations.

Experiment 4: Varying face sequences

Lastly, a fourth experiment observed how varying the number of face sequences affects runtime performance. In this experiment, increasing the number of face sequence increases the size of Γ_t proportionately. Specifically, the number of rows and columns both increase by a multiple of the number of face sequences. Thus, the duration of the *UWK* stage becoming longer as the time to invert Γ_t becomes longer. In addition, multiple face sequences must be processed by the CSRN network serially. This multiplies the time to perform the CSRN$_{\mathrm{FF}}$, CSRN$_{\mathrm{FB}}$, and *CCA* stages by the number of face sequences to process.

To evaluate the fourth experiment, the number of different face sequences per people class used during training varied from one to five. Also, five people classes, 49 samples per sequence, and 25 PCA components were used. Table 13 shows the results of this experiment. Also, Table 13 shows the size of Γ_t as the number of face sequences increased. Table 14 shows the resultant speedup performance.

Table 13. Training times using an increasing number of face sequences. The implementations shown are GCC single-core CPU (G-CPU), ICC single-core CPU (I-CPU), C1060 GPGPU, C2050 GPGPU, and C2070 GPGPU implementations using an increasing number of face sequences. In addition, the row×column size of Γ_t is shown as the number of face sequences increases.

Face Sequences	Γ_t Size	G-CPU (s)	I-CPU (s)	C1060 GPGPU (s)	C2050/C2070 GPGPU (s)
1	49×49	3,020.28	538.07	160.85	98.86
2	98×98	4,573.17	843.73	273.8	153.86
3	147×147	6,591.62	1,243.13	404.75	219.41
4	196×196	8,966.28	1,743.47	564.25	304.83
5	245×245	11,638.75	2,407.37	802.17	396.3

Table 14. Speedup comparison using runtimes shown in Table 13. The implementations shown are GCC single-core CPU (G-CPU), ICC single-core CPU (I-CPU), C1060 GPGPU, C2050 GPGPU, and C2070 GPGPU.

Face Sequences	Speedup over G-CPU			Speedup over I-CPU	
	I-CPU	C1060 GPGPU	C2050/C2070 GPGPU	C1060 GPGPU	C2050/C2070 GPGPU
1	5.61	18.78	30.55	3.35	5.44
2	5.42	16.7	29.72	3.08	5.48
3	5.3	16.29	30.04	3.07	5.67
4	5.14	15.89	29.41	3.09	5.72
5	4.83	14.51	29.37	3	6.07

The data shown in Table 13 and Table 14 reveal the advantage of the GPGPU implementations. When compared to both single-core CPU implementations, the GPGPU implementations continue to maintain a speedup advantage. Similar to previous results, the achieved speedup can be attributed to the parallel processing of the increasing CSRN_{FF} and CSRN_{FB} computations and the *CCA* and *UWK* vector/matrix operations.

In the case of the GCC compilation, the C1060 and C2050/C2070 GPGPU implementations show a decrease in the available speedup. This is likely due to the increasing amount of memory transfers. The increase in memory transfers result from the CPU needing to send additional face sequence input data to the GPGPU. Memory transfers between the CPU and GPGPU are very costly and decrease the amount of acceleration benefit seen. The additional parallel resources within the C2050/C2070 GPGPU implementation allows for a much slower decrease in acceleration when compared to the C1060 GPGPU implementation.

In the case of the ICC compilation, the C1060 GPGPU implementation continues to show this trend of decreasing speedup. However, the C2050/C2070 GPGPU shows a gradual increase in speedup. The ICC compilation has a much faster decrease in acceleration in comparison to the C2050/C2070 GPGPU implementation resulting in the gradual increase in speedup.

4.3.2.3 Multi-core and multi-GPGPU

Multi-core and multi-GPGPU implementations of the algorithm were explored. The main objective behind the multi-core and multi-GPGPU implementations was to observe how CSRN based face recognition scales as more CPU core/GPGPU resources are added. For these experiments only the ICC compilation results are shown as they are much faster than the GCC compilation.

Given that the Condor cluster is a heterogeneous cluster composed of three different kinds of GPGPUs (94 C2050, 14 C2070, and 48 C1060), this study utilizes the most optimal combinations possible when adding GPGPU resources. Since computations are bound by the C1060s, the C2050s and C2070s are scheduled prior to the C1060s to ensure optimal productivity. Specifically, the C2050s are added first and then the C2070s. At the time of these experiments, only 10 of the 14 C2070 were available. Therefore, after adding 10 C2070s, C1060s are added. The workload used for these experiments were one CSRN per primary to secondary process transmission.

Figure 30 shows the runtime performance of CSRN network designed to classify five people classes using five face sequences, 49 samples per face sequence, and 25 PCA components. For this CSRN network, Figure 30 shows the MSEKF training time decrease as the number of secondary processes increased from 10 to 150 by increments of 10. In the case of the multi-core implementation, the training time decreased from 254.26s to 20.58s using 10 to 150 secondary processes, respectively. For the multi-GPGPU implementation, the training time decreased from 43.08s to 9.07s using 10 to 150 secondary processes, respectively.

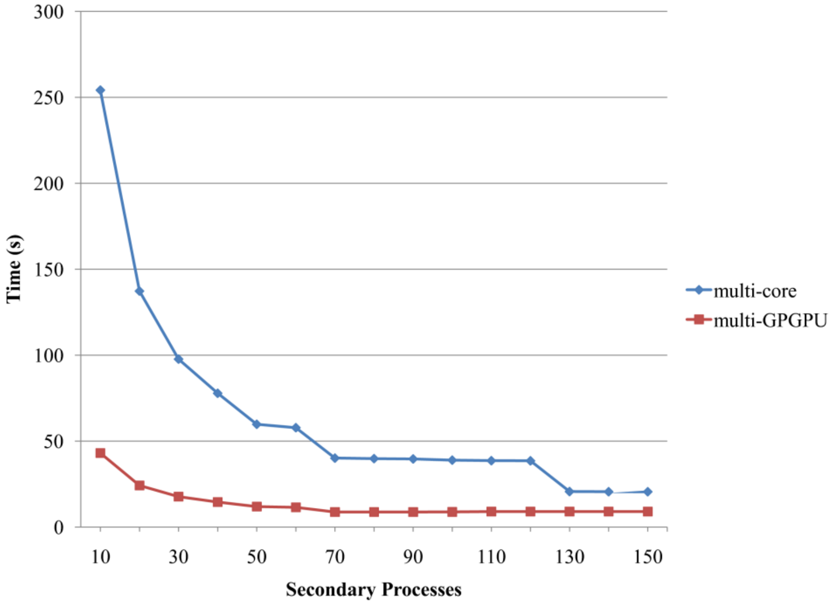


Figure 30. Graph of training time for multi-core and multi-GPGPU implementations.

Figure 31 shows the speedup for both the multi-core and multi-GPGPU implementations over the single-core ICC compiled CPU version. While both implementations offer vast improvement over the single-core version, the advantage of the multi-GPGPU implementation exceeds the multi-core. However, the multi-core implementation appears to scale better as the multi-GPGPU implementation levels off after 70 secondary processes. With the inclusion of the slower C1060 GPGPUs, the scaling of the multi-GPGPU implementation is hindered as expected. Once the number of secondary processes goes beyond the number of available C2050 (94) and C2070 (8) GPGPUs, the total efficiency of the multi-GPGPU implementation declines as the C1060 GPGPUs are used. This is shown by the decrease in speedup performance after 100 GPGPUs. As previously established, the C1060 GPGPUs processes data slower than the C2050 and C2070 GPGPUs for this application.

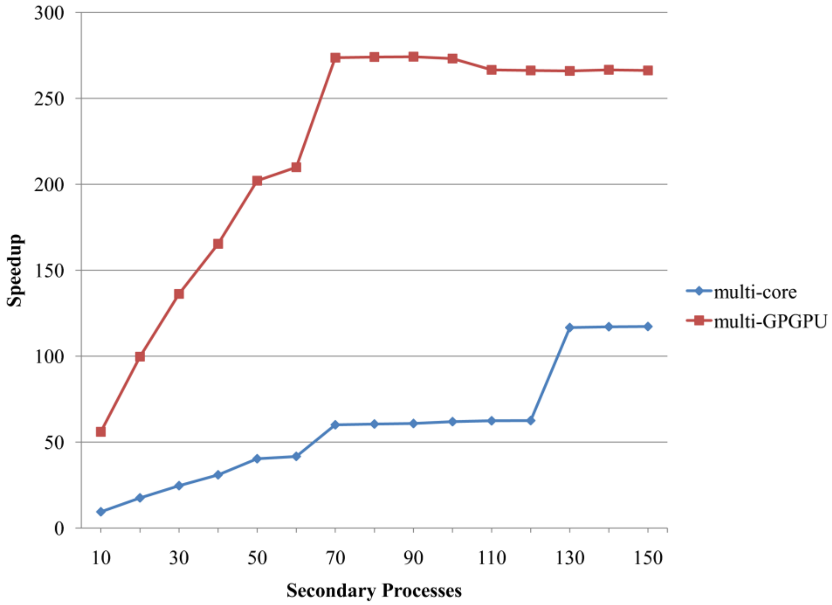


Figure 31. Graph of speedup for multi-core and multi-GPGPU implementations.

Table 15. Training runtime performance for multi-core and multi-GPGPU. The multi-core and multi-GPGPU implementations are compared to GCC (G-CPU), ICC (I-CPU), C1060 GPGPU, and C2050/C2070 GPGPU implementations.

Implementation	Time (s)	Speedup over G-CPU	Speedup over I-CPU
G-CPU	11,638.75	1	0.21
I-CPU	2,407.37	4.83	1
C1060 GPGPU	782.89	14.87	3.07
C2050/C2070 GPGPU	393.94	29.54	6.11
multi-core	20.58	565.44	116.96
multi-GPGPU	9.07	1,283.62	265.5

Table 15 shows the MSEKF training time and speedup performance for the multi-core and multi-GPGPU implementation to the previously discussed implementations (GCC and ICC single-core CPU implementations and the C1060 and C2050/C2070 single GPGPU implementations) for the experiment of five people classes, 49 samples per face sequence, five face sequences, and 25 PCA components. The timing results of the multi-core and multi-GPGPU implementations use 150 secondary processes. The speedup shown is a comparison between all implementations to both GCC and ICC single-core CPU implementations. From using the multi-GPGPU implementation, significant speedups of approximately 1,283.62 and 265.50 times are observed when compared against the GCC and ICC single-core CPU compilations respectively. Table 15 demonstrates the merits of the multi- GPGPU implementation.

4.3.2.4 Combining multi-core and multi-GPGPU

Given the limited GPGPU resources of the Condor cluster in comparison to the ample amount of Intel multi-core processors available, the computational capabilities of both multi-core and multi-GPGPU implementations need to combine to take greater advantage of cluster resources. Since each processor in the Condor cluster has six cores, only two cores per processor were used in the multi-GPGPU implementation. The remaining four cores were idle. To fully utilize each core in the system, the idle cores should perform CSRN computations concurrently with the GPGPUs. Using both multi-core and multi-GPGPU together, one can expect to achieve even greater runtime performance than using each alone.

For the multi-core, multi-GPGPU, and multi-core/GPGPU implementations, 75 of the total 78 multi-core processors on the system were utilized. In this fashion, 450 cores (one core for the primary process and 449 for the secondary processes) were used for the multi-core implementation. The multi-GPGPU implementation used 150 (94 C2050s, 8 C2070s and 48 C1060s) GPGPUs operating as secondary processes and one core as the primary process. Lastly, the multi-core/GPGPU implementation used one core as the primary process and 299 cores as secondary processes. The remaining cores 150 cores used GPGPUs as secondary processes. As before, the multi-core implementations were compiled using ICC.

Also, more resources of the Condor cluster should be utilized to get an idea of the Condor cluster's computational capability for CSRN based face recognition. Therefore, two additional experiments were conducted to demonstrate the significant impact on runtime performance a system such as the Condor cluster has for this application. Additionally, the runtime performance of the the multi-core, multi-GPGPU,and multi-core/GPGPU implementations are compared. In these experiments, the number of samples per face sequence and number of face sequences were varied.

Experiment 5: Varying samples per face sequence using large networks

This experiment observed the effect varying the number of samples per face sequence has on runtime performance for the multi-core, multi-GPGPU, and multi-core/GPGPU implementations. For this experiment, a randomly generated CSRN network to classify 1,000 people classes using 10 PCA components and one face sequence per people class was created. The number of times the face sequence was sampled varied using the following sample rates: 25, 36, 49, 81, 100, 121 and 144. This resulted in a full network of 10,000 CSRNs. Figure 32 shows the training time results of this experiment, and Figure 33 shows the speedup performance over the ICC single-core compilation.

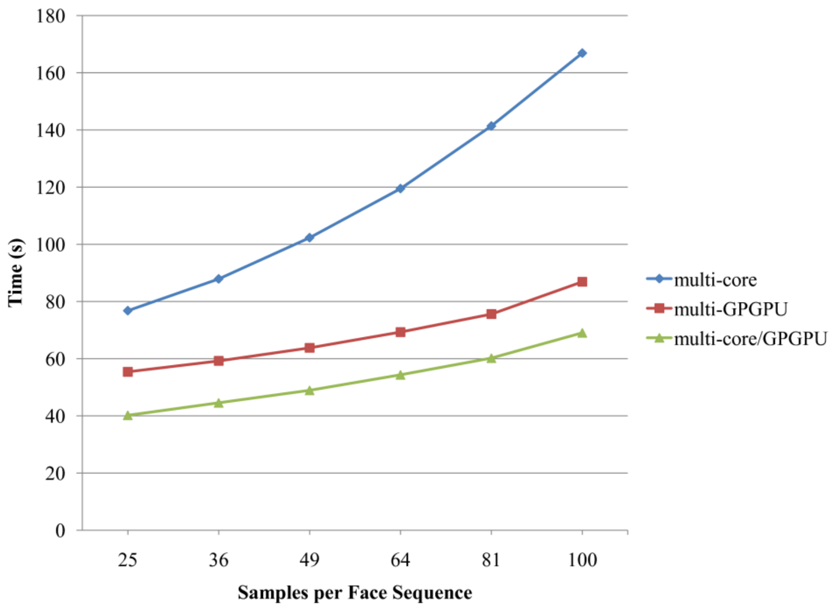


Figure 32. Graph showing varying number of samples per face sequence. Shows the training timing for the ICC compiled multi-core, multi-GPGPU, and multi-core/GPGPU.

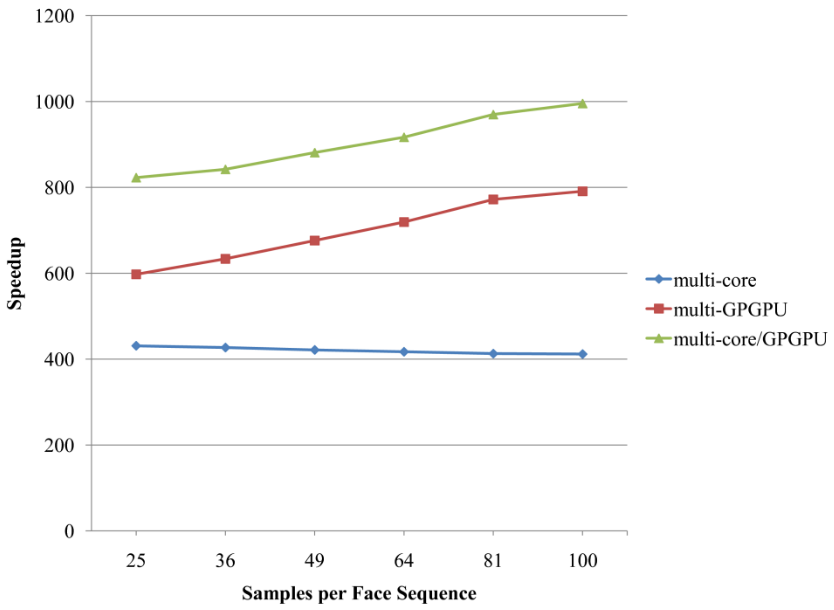


Figure 33. Speedup graph for varying the number of samples per face sequence. Shows the speedup performance for the ICC compiled multi-core, multi-GPGPU, and multi-core/GPGPU.

As seen in Figure 32 and Figure 33, both implementations incorporating GPGPUs are faster than the multi-core implementation as the number of samples per face sequence increases. The multi-core implementation speedup is shown to decrease as face sequence sample increase. The multi-GPGPU and multi-core/GPGPU implementations speedup performance improves while the samples per face sequence increase.

Another result of significance is that the multi-GPGPU implementation is faster than the multi-core implementation using much less resources. The multi-GPGPU implementation uses 150 GPGPU to act as secondary processes compared to the 449 cores that the multi-core implementation uses. As expected, the multi-core/GPGPU implementation is faster than the multi-GPGPU implementation as the the former uses the additional processing power of cores during computation. As seen in Figure 33, the multi-core/GPGPU implementation displays a speedup performance of approximately 823 to 996 times for 25 to 100 samples per face sequence, respectively.

Experiment 6: Varying face sequences using large networks

This experiment observed the behavior of the multi-core, multi-GPGPU, and multi-core/GPGPU implementations while varying the number of face sequences from one to five. To do this, another randomly generated 1,000 people class CSRN network using 10 PCA components was created. The face sequences used in this network were sampled 25 times. Figure 34 and Figure 35 show the training time results and speedup performance of the ICC compiled multi-core, multi-GPGPU, and multi-core/GPGPU implementations, respectively. The speedup performance shown in Figure 35 uses the ICC single-core compilation as a base.

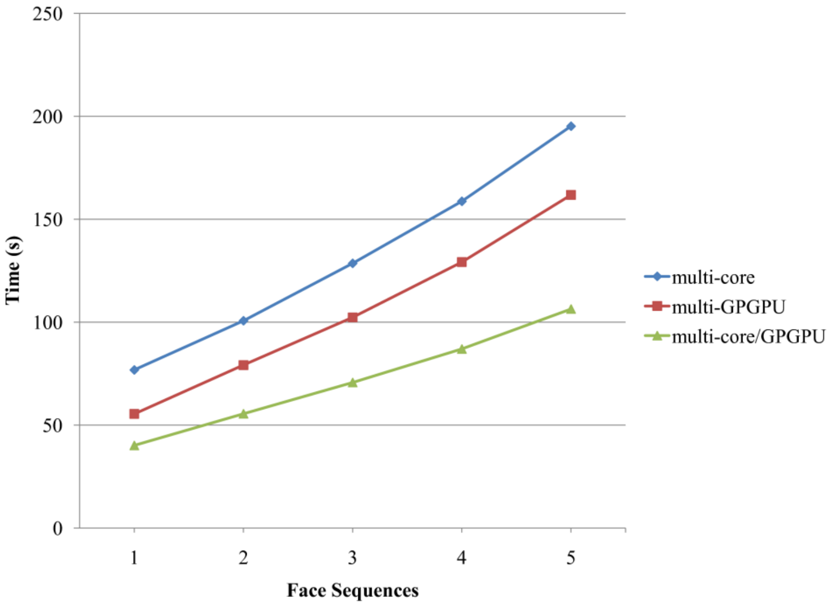


Figure 34. Graph of varying the number of face sequences. Shows the training timing for the ICC multi-core, multi-GPGPU, and multi-core/GPGPU implementations.

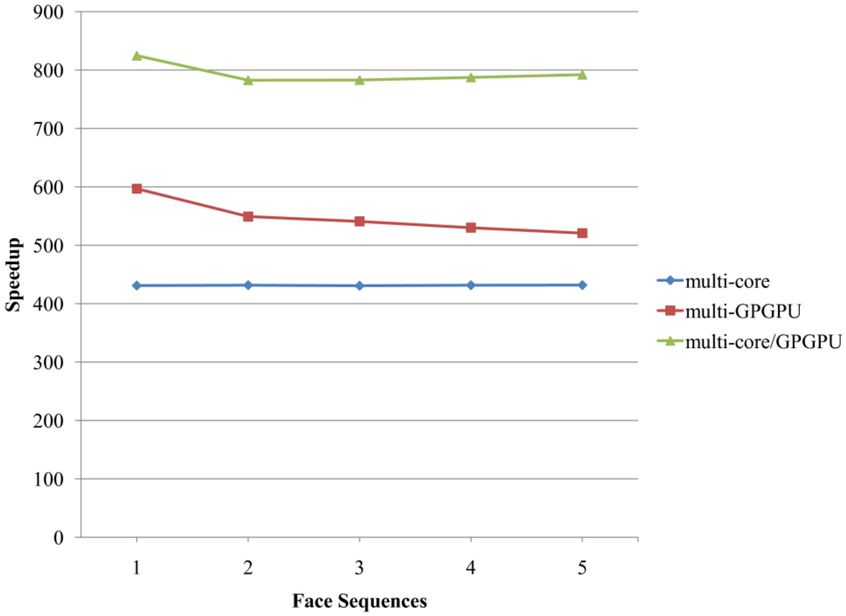


Figure 35. Speedup graph for varying the number of face sequences. Shows the speedup performance for the ICC multi-core, multi-GPGPU, and multi-core/GPGPU implementations.

In Figure 34 like the previous experiments, the multi-GPGPU implementation maintains a lower training runtime when compared to the multi-core. As before, the multi-core/GPGPU implementation achieved the lowest runtime time. In Figure 35, the speedup performance of the multi-core implementation remains relatively fixed at approximately 431 times. While higher than the multi-core, the speedup performance of the multi-GPGPU gradually decreases from 596 times to 520 times as the number of face sequences increases from one to five. As expected, the speedup performance of the multi-core/GPGPU is greater than both multi-core and multi-GPGPU as it fluctuates between 824 and 792 times as the number of face sequences increase. This fluctuation is likely the result of load balancing the decreasing GPGPU processing times with the relatively stable multi-core processing times.

4.4 RBFNN Acceleration

The training and testing data used for the experimentation were acquired using a CompuGauge tool [54]. The training data consisted of 1989 position data points and the testing dataset contained 192 positions points (separate and independent datasets). The objective of our RBF network is to map the *actual* position data points/coordinates to that of the corresponding *desired* position data points/coordinates. Our goal is to find the optimal spread value of the non-linear activation functions (i.e., Radial Basis Functions). As a result, we train the network using different spread values (range: 100 to 5000 in steps of 50). For a specific spread value, we perform a 7-fold cross validation analysis [55]. The performance of the network is then computed by examining the difference between the network's output and the corresponding *desired* data (i.e. error). Based on such error information, the optimal spread value is found to be 3200 (spread value yielding minimum error). Note that the optimal spread value is obtained using only the training dataset. The performance of the network (i.e., error discussed in the latter) versus the range of increasing spread values is depicted in Figure 36. The curve is generated using a 7-fold cross validation performance analysis working on the training data *only*. As expected the error decreases as the spread value increases. It is important to note that after a certain threshold/spread-value there is no significant performance improvement. Also note that as the spread value gets extremely large, the distance matrix approaches a singularity (i.e., the matrix inversion process appears unstable for larger spread values).

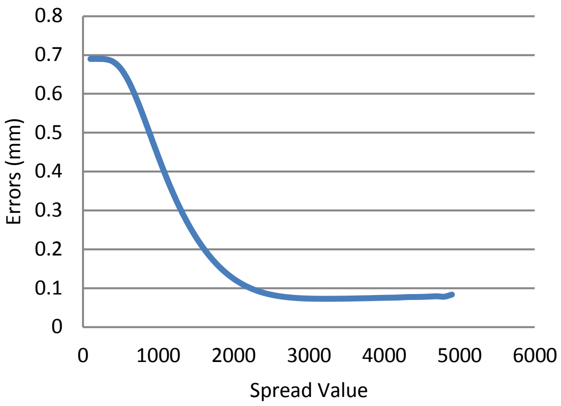
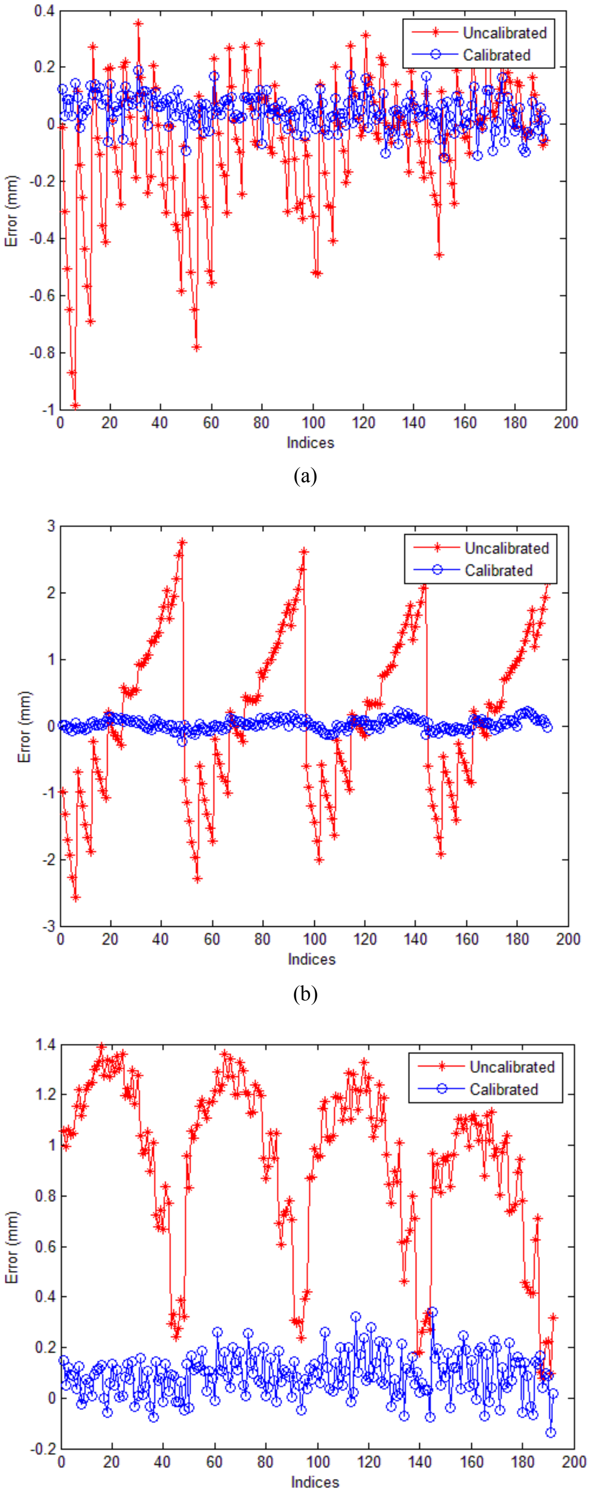


Figure 36. Performance/error versus increasing spread values.

After completing the training process, we validated the robustness of the network by testing it on the separate and independent testing dataset that consisted of the 192 position points. Figure 37 shows the testing performance results. The figure shows the performance in terms of error (i.e., difference between *desired output* and the actual network's output) versus testing point indices. Figure 37a shows the error/performance for the *x*-dimension. Figure 37b and Figure 37c show the performance results for *y* and *z* dimensions respectively. The figure also shows two curves, one illustrating the discrepancy between the *desired* and *measured/actual* position data points (uncalibrated), and the other showing performance after employing the RBF network (i.e., calibrated/filtered performance). Based on Figure 37, we can safely assume that the RBF-network/filter did indeed improve the accuracy of the MA1400 MOTOMAN manipulator (the network/filter has significantly reduced the original discrepancy between the *desired* and *actual* data). Table 16 shows the average testing error(s). As shown in the table, the average error before calibration is 0.706 mm and the average error after calibration is 0.076 mm (or 76 microns). Our approach shows an accuracy/precision improvement in the order of 88%.



(a)

(b)

(c)

Figure 37. Performance results (error vs. testing point indices): (a) error in the *x*-dimension, (b) error in the *y*-dimension, (c) error in the *z*-dimension.

The initial implementation of this algorithm was in MATLAB R2010 installed on a Core Duo 8600, 4-GB memory desktop computer. It required about one week to train the dataset described. The GPGPU implementation of the algorithm ran in about 10 minutes, corresponding to a speedup of about 300 times. Figure 38 depicts the speedup performance when comparing the two platforms.

Table 16. Calibration results (average error[s]/performance).

	x (mm)	*y (mm)*	*z (mm)*	***Overall Average Error***
Average error before calibration	0.177±0.171	1.005±0.646	0.936±0.326	0.706
Average error after calibration	0.062±0.04	0.067±0.049	0.1±0.068	0.076

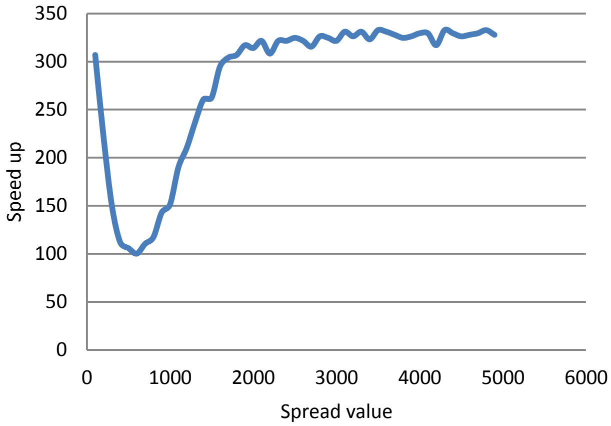


Figure 38. Speed up performance (GPGPU versus MATLAB).

5. Conclusion

This study examined the parallelization and acceleration of several different neuromorphic computing algorithms using a variety of multicore architectures. The results indicate that these algorithms have a high degree of parallelism and are well suited multicore architectures. Some of the models exhibit high degrees of data level parallelism and so are suitable for acceleration on GPGPU platforms. These are the neural network based models (SNN, CSRN, and RBFNN). The Bayesian network based models (HTM and Dean model) do have significant thread level parallelism but not much data level parallelism. Thus they are well suited to non-SIMD multicore platforms (such as Intel Xeon processors) but not suited for GPGPU like platforms. All the algorithms examined scaled well on multiprocessor computing clusters.

The following list publications that have resulted directly from this study:

- P. Yalamanchili and T. M. Taha, “Multicore Cluster Implementations of Hierarchical Bayesian Cortical Models,” International Conference on Computer and Information Technology (ICCIT), December 2009.
- M. A. Bhuiyan, V. K. Pallipuram, M. C. Smith, T. M. Taha, R. Jalasutram, "Acceleration of spiking neural networks in emerging multi-core and GPU architectures," IEEE International Symposium on Parallel & Distributed Processing, April 2010.

- B. Han and T. M. Taha, “Acceleration of spiking neural network based pattern recognition on NVIDIA graphics processors,” *Applied Optics*, 49:(101), 83-91, April 2010.
- T. M. Taha, P. Yalamanchili, M. Bhuiyan, R. Jalasutram, C. Chen, and R. Linderman, "Neuromorphic Algorithms on Clusters of PlayStation 3s," International Joint Conference on Neural Networks, July 2010.
- B. Han and T. M. Taha, “Neuromorphic Models on a GPGPU Cluster,” International Joint Conference on Neural Networks, July 2010.
- P. Yalamanchili, S. Mohan, R. Jalasutram, and T. M. Taha, “Acceleration of Hierarchical Bayesian Network Based Cortical Models on Multicore Architectures,” *Parallel Computing,* 36:(8), 449-468, August 2010.
- K. Rice, T. M. Taha, R. Miller, K. M. Iftekharuddin, K. Anderson, and T. Salan, “Accelerating CSRN based face recognition on an NVIDIA GPGPU,” Infotech@Aerospace Conference, March 2011 (Invited paper).
- T. Messay, C. Chen, R. Ordóñez and T. M. Taha, “GPGPU Acceleration of a Novel Calibration Method for Industrial Robots,” IEEE National Aerospace & Electronics Conference, July 2011.
- K. L. Rice, T. M. Taha, K. M. Iftekharuddin, K. Anderson, and T. Salan, “GPGPU Acceleration of Cellular Simultaneous Recurrent Networks Adapted for Maze Traversals,” IEEE International Joint Conference on Neural Networks (IJCNN), August 2011.
- T. Taha and C. Chen, “Spiking Neural Networks on High Performance Compute Clusters,” SPIE Optics and Photonics for Information Processing V, August 2011.
- K. Rice, T. M. Taha, K. M. Iftekharuddin, K. Anderson, and T. Salan, “Multicore Cluster Acceleration of Cellular Simultaneous Recurrent Network Based Face Recognition,” *International Journal of Parallel Programming,* to be submitted.
- C. Chen, T. M. Taha, M. Bhuyian, and R. Jalasutram, “Acceleration of Spiking Neural Network Based Character Recognition Models on Multicore Architectures,” *IEEE Transactions on Parallel and Distributed Systems,* to be submitted.
- T. Messay, C. Chen, R. Ordóñez and T. M. Taha, “Hardware Accelerated Calibration for Industrial Robots,” *IEEE Transactions on Robotics*, to be submitted.

The insights gained from this study using existing multicore platforms are currently being utilized by Dr. Taha’s team to develop novel multicore architectures for neuromorphic algorithms. In particular we are examining the design of novel memristor [56] based computing platforms through detailed circuit and system level simulations. Our studies have shown potential for over a 1000 times speedup over the latest Intel Xeon processors, while utilizing a fraction of the power of the Xeon processors [57]. Such systems have broad military applications, including new lightweight, low power, intelligent devices for use by soldier on the battleground.

In these follow-on studies we utilized detailed memristor SPICE models we developed [58] and memristor circuits we designed [59], to design each computing core. These follow-on studies would not be possible without the insights gained from the AFOSR funded study outlined in this report. The final conclusion for this study would be that neuromorphic algorithms are well suited to parallelization on mulicore architectures, and that significant scope exists for the design of even more specialized multicore architectures to achieve higher performances. Given the broad range of military applications for these algorithms (in data mining, data fusion, etc), further study is needed into the design of such novel architectures.

References

[1] C. Gao and D. Hammerstrom, “Cortical Models onto CMOL and CMOS—Architectures and Performance/Price,” *IEEE Transactions on Circuits and Systems I*, 54(11):2502–2515, Nov 2007.

[2] C. Johansson and A. Lansner, “Towards Cortex Sized Artificial Neural Systems,” *Neural Networks*, 20(1), 48–61, Jan. 2007.

[3] W. Rall, “Branching dendritic trees and motoneuron membrane resistivity,” *Experimental Neurology*, 1, 503–532, 1959.

[4] I. Segev, and W. Rall, “Excitable dendrites and spines: earlier theoretical insights elucidate recent direct observations,” *Trends in Neuroscience*, 21, 453–460, 1998.

[5] A. L. Hodgkin and A. F. Huxley, “A quantitative description of membrane current and application to conduction and excitation in nerve,” *Journal of Physiology*, 117, 500–544, 1952.

[6] W. Maas, “Networks of spiking neurons: the third generation of neural network models,” *Transactions of the Society for Computer Simulation International*, 14(4), 1659–1671, Dec. 1997.

[7] A. Delorme and S. J. Thorpe, “SpikeNET: an event-driven simulation package for modelling large networks of spiking neurons,” *Network-computation in neural systems*, 14(4), 613–627, Nov. 2003.

[8] S. Zhu and D. Hammerstrom, “Simulation of associative neural networks,” *Proceedings of the 9th International Conference on Neural Information Processing*, 1639- 1643, Nov. 2002.

[9] R. Ananthanarayanan and D. Modha, “Anatomy of a Cortical Simulator,” *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Supercomputing 2007), Reno, NV, November 2007.

[10] M. Djurfeldt, M. Lundqvist, C. Johansson, M. Rehn, O. Ekeberg, and A. Lansner, “Brain-scale simulation of the neocortex on the IBM Blue Gene/L supercomputer,” *IBM Journal of Research and Development*, 52(1-2), 31–41, Jan.-Mar. 2008.

[11] E. Izhikevich and G. Edelman, "Large-Scale Model of Mammalian Thalamocortical Systems," Proceedings of the National Academy of Sciences, 105(9), 3593–3598, Mar. 2008.

[12] V. B. Mountcastle, “An organizing principle for cerebral function: The unit module and the distributed system,” *The Neurosciences Fourth Study Program* (F. O. Schmitt and F. G. Worden, Eds.). Cambridge: MIT Press, pp. 21–42, 1979.

[13] V. B. Mountcastle, “Introduction to the special issue on computation in cortical columns,” *Cerebral Cortex*, 13(1):2–4, 2003.

[14] O. Sporns and J. D. Zwi, “The small world of the cerebral cortex,” *Neuroinformatics*, 2(2), 145–162, 2004.

[15] M. Newman, D. Watts, and S. Strogatz, “Random graph models of social networks,” *Proceedings of the National Academy of Science*, 2566–2572, 2002.

[16] T. Dean, “A Computational Model of the Cerebral Cortex,” *Proceedings of the Twentieth National Conference on Artificial Intelligence* (AAAI-05), 2005.

[17] J. A. Anderson, “Arithmetic on a parallel computer: Perception versus logic,” *Brain and Mind*, 4, 169–188, 2003.

[18] R. Zemel, “Cortical belief networks,” in Hecht-Neilsen, R., ed., *Theories of the Cerebral Cortex,* New York, NY: Springer-Verlag, 2000.

[19] J. Hawkins and D. George, “Hierarchical Temporal Memory – Concepts, Theory, and Terminology,” Whitepaper, Numenta Inc, May 2006.

[20] T. S. Lee and D. Mumford, “Hierarchical bayesian inference in the visual cortex,” *Journal of the Optical Society of America A*, 2(7), 1434–1448, 2003.

[21] T. Dean, “Learning invariant features using inertial priors,” *Annals of Mathematics and Artificial Intelligence*, 47(3-4), 223–250, Aug. 2006.

[22] D. George, “How the brain might work: A hierarchical and temporal model for learning and recognition,” Doctoral Dissertation, Dept. of Electrical Engineering, Stanford University, 2008.

[23] Y. Ren, K. M. Iftekharuddin, and W. E. White, “Large-scale pose-invariant face recognition using cellular simultaneous recurrent network”, Applied Optics, vol. 49, no. 10, pp. B92-B103, 2010.

[24] T. Messay, C. Chen, R. Ordóñez and T. M. Taha, “GPGPU Acceleration of a Novel Calibration Method for Industrial Robots,” IEEE National Aerospace & Electronics Conference, July 2011.

[25] T. Dean. “Scalable inference in hierarchical generative models,” Ninth International Symposium on Artificial Intelligence and Mathematics, 2006.

[26] E. M. Izhikevich, “Simple Model of Spiking Neurons,” IEEE Trans. Neural Networks, vol. 14, pp. 1569-1572, Nov. 2003.

[27] A. L. Hodgkin and A. F. Huxley, “A quantitative description of membrane current and application to conduction and excitation in nerve,” Journal of Physiology, vol. 117, pp. 500–544, 1952.

[28] C. Morris and H. Lecar, “Voltage oscillations in the barnacle giant muscle fiber,” Biophys. J., vol. 35, pp. 193–213, 1981.

[29] H. R. Wilson, “Simplified dynamics of human and mammalian neocortical neurons,” J. Theor. Biol., vol. 200, pp. 375–388, 1999.

[30] E. M. Izhikevich., “Dynamical Systems in Neuroscience”, MIT press, Cambridge, Massachusetts, 2007

[31] W. Gerstner, W. Kistler, “Spiking Neuron Models, Single neurons, Populations, Plasticity,” Cambridge University Press, 2002

[32] E.Izhikevich, “Which Model to Use for Cortical Spiking Neurons?” *IEEE Transactions on Neural Networks*, 15(5), 1063-1070, 2004.

[33] H. Markram, “The Blue Brain Project,” *Nature Reviews Neuroscience*, 7, 153–160, 2006.

[34] D. George and J. Hawkins, “A hierarchical Bayesian model of invariant pattern recognition in the visual cortex,” *Proceedings of the International Joint Conference on Neural Networks*, 2005.

[35] J. Hawkins and S. Blakeslee, “On Intelligence,” Times Books, Henry Holt and Company, New York, NY 10011, Sept. 2004.

[36] J. Pearl, “Probabilistic Reasoning in Intelligent Systems,” *Networks of Plausible Inference*, Morgan Kaufmann, San Francisco, CA, 1988.

[37] T. Dean. “Scalable inference in hierarchical generative models,” Ninth International Symposium on Artificial Intelligence and Mathematics, 2006.

[38] T. S. Lee and D. Mumford, “Hierarchical bayesian inference in the visual cortex,” *Journal of the Optical Society of America A*, 2(7), 1434–1448, 2003.

[39] S. Lauritzen, D. Spiegelhalter, “Local computations with probabilities on graphical structures and their application to expert systems,” *Journal of the Royal Statistical Society*, 50(2):157–194, 1988.

[40] R. Ilin, R. Kozma, and P. J. Werbos, “Beyond feedforward models trained by backpropagation: A practical training tool for a more efficient universal approximator,” *IEEE Transactions on Neural Networks*, vol. 19, no. 6, pp. 929–937, 2008.

[41] K. M. Iftekharuddin and K. Anderson, “Image registration under affine transformation using cellular simultaneous recurrent networks,” *SPIE Optics and Photonics for Information Processing IV*, vol. 7797, no. 1, p. 77970, 2010.

[42] Y. Ren, K. M. Iftekharuddin, and W. E. White, “Large-scale pose-invariant face recognition using cellular simultaneous recurrent network,” *Applied Optics*, vol. 49, no. 10, pp. B92–B103, 2010.

[43] B. Mooring, M. Driels, and Z. Roth, *Fundamentals of Manipulator Calibration*, John Wiley & Sons, Inc. New York, NY, USA, 1991.

[44] “NX100 Controller Manual”, Part Number:149201-1, Yaskawa Motoman Robotics Inc., December 19, 2007.

[45] S. Vassiliadis, S. Cotofana, P. Stathis “Block based compression storage expected performance”, in the proceedings of HPCS2000, 2000.

[46] Thinking in CUDA: Gauss-Jordan elimination. [Online] Available at: http://cudahacks.com/cms/Blog/tabid/64/EntryId/7/Thinking-in-CUDA-Gauss-Jordan-elimination.aspx.

[47] J. Choi., J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "The Design and Implementation of the ScaLAPACK LU, QR and Cholesky Factorization Routines," *Scientific Programming*, Vol. 5, No. 3, 173-184, Aug. 1996.

[48] V. Volkov and J. Demmel, "LU, QR and Cholesky Factorizations usingg Vector Capabilities of GPUs", Technical Report of UC Berkely, May 2008.

[49] NVIDIA Corp., *NVIDIA CUDA Programming Guide 4.0*, March, 2011.

[50] G. Bradski and A. Kaehler, Learning OpenCV:Computer Vision with the OpenCV Library. Cambridge, MA: O'Reilly Press, 2008.

[51] NVIDIA. NVIDIA Developer Zone. [Online] Available at: http://developer.nvidia.com/object/cuda_3_2_downloads.html.

[52] NVIDIA. NVIDIA CUDA C Programming Guide, version 3.2, 2010. [Online] Available at: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf .

[53] D. B. Graham and N. M. Allinson, "Characterizing Virtual Eigensignatures for General Purpose Face Recognition", in Face Recognition: From Theory to Applications, NATO ASI Series F, Computer and Systems Sciences, vol. 163, H. Wechsler, P. J. Phillips, V. Bruce, F. Fogelman-Soulie, and T. S. Huang Eds., 1998, pp. 446-456. [Online] Available at: http://www.sheffield.ac.uk/eee/research/iel/research/face

[54] *3-D COMPUGAUGE Manual (version 4.0).* Compugauge: Dynalog Inc., (http://www.dynalog-us.com/), 2002.

[55] K. Fukunaga, *Introduction to Statistical Pattern Recognition*, Academic Press, Inc., 1990.

[56] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing Memristor found," *Nature*, 453, 80–83 (2008).

[57] T. Taha, M. R. Hasan, C. Yakopcic, M. McLean, "Towards Memristor Based Neuromorphic Computing Architectures," to be submitted to the IEEE International Joint Conference on Neural Networks, 2013.

[58] C. Yakopcic, T. M. Taha, G. Subramanyam, R. E. Pino, S. Rogers, "A Memristor Device Model," *IEEE Electron Device Letters*, Oct. 2011.

[59] C. Yakopcic, T. M. Taha, G. Subramanyam, and S. Rogers, "Multiple Memristor Read and Write Circuit for Neuromorphic Applications," *International Joint Conference on Neural Networks (IJCNN),* (2011).